

EIN KONZEPT ZUR INTERAKTIVEN PROGRAMMIERUNG VON
MIKRORECHNERN IN REAL-TIME BASIC

Dissertation
zur Erlangung des Grades eines
Doktor-Ingenieurs
der Abteilung für Elektrotechnik
an der Ruhr-Universität Bochum

von

WERNER HEINZEL

Bochum

Bochum 1976

EIN KONZEPT ZUR INTERAKTIVEN PROGRAMMIERUNG VON
MIKRORECHNERN IN REAL-TIME BASIC

Dissertation
zur Erlangung des Grades eines
Doktor-Ingenieurs
der Abteilung für Elektrotechnik
an der Ruhr-Universität Bochum

von

WERNER HEINZEL

Bochum

Bochum 1976



Dissertation eingereicht am: 5. Mai 1976

Tag der mündlichen Prüfung: 22. Juni 1976

Hauptreferent: Prof.Dr.-Ing. W.Weber

Korreferent: Prof.Dr.-Ing. F.Depping

INHALTSVERZEICHNIS

	Seite
ZUSAMMENFASSUNG	6
1. PROBLEMSTELLUNG	8
1.1 Einleitung	8
1.2 Zielsetzung	10
2. MÖGLICHKEITEN DER SOFTWAREERSTELLUNG BEI MIKRORECHNERN	11
2.1 Eigenschaften der Mikrorechnersysteme	11
2.2 Softwareunterstützung bei der Erstellung von Programmen	13
2.3 Grenzen der derzeitigen Programmierverfahren bei Mikro- prozessoren	16
3. EIN ÜBERSETZERKONZEPT FÜR MIKRORECHNER	19
3.1 Forderungen an einen anwenderfreundlichen Übersetzer für Mikrorechner	19
3.1.1 Benutzerfreundlichkeit	19
3.1.2 Prozeßanforderung	19
3.1.3 Hardwarebegrenzungen	20
3.2 Konzept eines Übersetzers für Mikrorechner	20
4. AUSWAHL EINER GEEIGNETEN PROGRAMMIERSPRACHE	24
4.1 Betrachtung bekannter Programmiersprachen	24
4.2 Real-Time BASIC	25
4.3 BASEX für Mikrorechner: BASEM	27
4.3.1 Konzept durchzuführender Modifikationen	27
4.3.2 Implementierter Sprachumfang	32
4.3.3 Syntax der BASEX-Modifikation BASEM	44

	Seite
5. IMPLEMENTIERUNG VON STANDARD-SYSTEMPROGRAMMEN	53
5.1 Arithmetische Unterprogramme	53
5.1.1 Umfang der zu realisierenden Unterprogrammbibliothek	53
5.1.2 Auswahl des Zahlenformates	55
5.1.3 Struktur der Unterprogramme und Organisation des Unterprogrammaufrufes	58
5.1.4 Verwendung der Unterprogrammbibliothek im "immediate mode"	60
5.2 Prozeßobjekte	62
5.3 Real-Time Systemvariable	65
6. EIN INTERAKTIVER ÜBERSETZER FÜR MIKRORECHNER	66
6.1 Struktur des Übersetzers	66
6.2 Lexikalische und syntaktische Analyse einer Quellenzeile	69
6.2.1 Die Eingabephase	69
6.2.2 Tabellenorganisation	73
6.2.3 Interaktive Analyse	78
6.3 Codegenerierung	85
6.3.1 Vorbemerkungen	85
6.3.2 Bearbeitung von Ausdrücken	86
6.3.3 Fehlerbehandlung	91
7. EIN BINDER FÜR ABSOLUT ADRESSIERTE PROGRAMME	92
7.1 Funktionsgerechte Verschiebung absolut adressierter Programme	92
7.2 Der Binder (linkage editor)	96
8. DAS KOPPLUNGSSYSTEM	99

	Seite
9. DAS KOMMANDOSYSTEM	102
9.1 Organisation	102
9.2 Aufbereitung von Quellenprogrammen	104
9.3 Übersetzung, Test und Bindevorgang	105
9.4 Sicherung erstellter Programme	108
9.5 Weitere Programmierhilfen	109
LITERATUR	111
ANHANG	122
LEBENS LAUF	130

ZUSAMMENFASSUNG

Die vorliegende Arbeit behandelt ein Konzept zur interaktiven Programmierung von Mikrorechnern in einer höheren Programmiersprache. Es wird ein dialogfähiges Übersetzersystem vorgeschlagen, welches keinen zusätzlichen Hintergrundspeicher erfordert und als Ein-/Ausgabegerät nur eine Standard-Fernschreibmaschine benötigt.

Die Programmerstellung in einer höheren Programmiersprache ermöglicht insbesondere Anwendern mit weniger guten Systemkenntnissen eine weitgehend selbständige Arbeitsweise. Die Auswahl der zu implementierenden Programmiersprache stellt deshalb eine wichtige Entscheidung dar.

Aus Standardisierungsgründen ist die Modifizierung einer bereits eingeführten und somit dem Anwender oft bekannten Programmiersprache der Einführung einer vollständig neuen Sprachkonstruktion vorzuziehen. Deshalb wurde als Grundlage der Implementierung einer höheren Programmiersprache für Mikrorechner die Sprache BASIC bzw. die Real-Time Erweiterung BASEX ausgewählt und im Hinblick auf die Anwendung bei Mikrorechnern so ergänzt bzw. modifiziert, daß u.a. eine erhebliche Rechenzeiterparnis möglich ist.

Das angegebene Übersetzersystem arbeitet vollständig interaktiv. Programmierfehler werden nach Eingabe jeder Quellenzeile unter Verwendung eines einfachen Analyseverfahrens erkannt und unmittelbar angezeigt. Zur Fehlerkorrektur genügt die erneute Eingabe der berichtigten Zeile. Die Codegenerierung erfolgt operatorprioritätsgesteuert unter Einsatz von Prioritätsfunktionen.

Das Übersetzersystem läßt sich mit bereits eingesetzten Mikrorechnern über die serielle Schnittstelle einer Fernschreibmaschine koppeln. Programmtests sind dann direkt unter Verwendung der vorhandenen Prozeßinterfaces möglich. Die interaktive Arbeitsweise gestattet die sofortige Programmkorrektur.

Zur weiteren Unterstützung des Anwenders dient ein Kommandosystem, welches neben den Leistungen sogenannter Grundkommandos, eine Anzahl von Programmierhilfen ermöglicht.

Das Übersetzerkonzept wurde am Beispiel des Mikroprozessors *Intel 8080* implementiert. Da für Mikroprozessoren noch keine Programmbibliothek verfügbar war, wird die Vorgehensweise zur Implementierung von Standard-Systemprogrammen bei Mikrorechnern angegeben und Ergebnisse genannt.

Der Übersetzer erzeugt von dem eingegebenen Quellenprogramm einen absolut adressierten Maschinencode. Vor Übergabe des Maschinenprogramms an den Mikrorechner müssen die einzelnen Programmelemente gebunden werden. Es wird ein Konzept angegeben, das die Abwicklung des Bindevorganges bei absolut adressierten Programmen gestattet.

1. PROBLEMSTELLUNG

1.1 Einleitung

Die Steuerung und Regelung von Prozessen wurde in den vergangenen Jahren meist einem einzigen Prozeßrechner übertragen, der als zentrales Rechensystem eine Vielzahl von Teilaufgaben zu koordinieren und zu lösen hatte [22]. Eine Abkehr von diesem zentralen Systemkonzept bedeutet Delegation anfallender Aufgaben in eine entsprechende Zahl von Kleinrechnern, also die Aufteilung der erforderlichen Rechnerkapazität. Durch dieses Konzept wird

- bessere Überschaubarkeit bei der Planung und Inbetriebnahme,
- größere Flexibilität,
- geringeres Investitionsrisiko,
- größere Sicherheit und
- erhöhte Verfügbarkeit des Gesamtsystems

erreicht [14]. Als Kleinrechner verwendete man bisher meist sogenannte 'Minirechner'. Dies sind in ihrem Leistungsumfang begrenzte aber kostengünstigere Rechensysteme als die größeren Prozeßrechner, die jedoch noch bei umfangreichen Prozessen zur Koordination der einzelnen Systeme als Leit- und Führungsrechner eingesetzt werden. Die meisten Minirechner weisen eine relativ umfangreiche Softwareunterstützung auf. So steht u.a. jeweils mindestens ein Compiler für eine höhere Programmiersprache zur Verfügung [55].

Durch fortschreitende Verbesserung des Integrationsgrades in der Halbleitertechnologie konnte seit 1971 in der Prozeßrechentechnik ein neues Bauelement eingesetzt werden: Der 'Mikroprozessor'. Es handelt sich um eine vollständige Rechnerzentraleinheit (CPU). Der Begriff Mikroprozessor bezieht sich dabei auf die Kleinheit der Baugruppe. Ein vollständiges Rechensystem - man spricht von einem 'Mikrorechner' - ergibt sich durch Umschaltung des Mikroprozessors mit den, für eine Rechnerstruktur erforderlichen weiteren Bauelementen nach Abb. 1. Die Mikroprozessoren wurden in

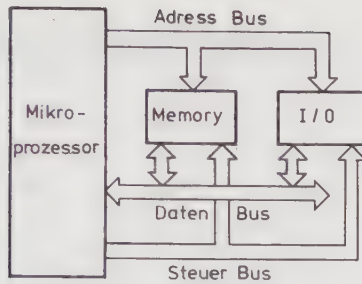


Abb. 1: Mikrorechner [12]

der Folgezeit stetig weiterentwickelt, so daß eine Anpassung der hardwaremäßigen Leistungsfähigkeit entsprechender Mikrorechner an die der Minirechner bei wesentlich geringerem Preis zu erwarten ist. Außer den Single-Chip-Mikroprozessoren kennt man auch Multi-Chip-Mikroprozessoren und mikroprogrammierbare Prozessoren. In [4] werden von FARBER die oben genannten verschiedenen Mikroprozessor-Strukturen untersucht, ihr Leistungsbereich gegenüber Minirechnern abgegrenzt und verschiedene Anwendungsfälle in der Prozeßrechner-technik vorgeschlagen. So können viele Aufgaben, die bisher den Minirechnern vorbehalten waren, ohne weiteres von den Mikrorechnern bewältigt werden [4,6]. Auch wurden verschiedenartige Techniken erst durch den Einsatz von Mikroprozessoren einem größeren Anwenderkreis zugänglich. Es sei an dieser Stelle nur an die 'freiprogrammierbaren Steuerungen' [2] oder 'Regelrechner' [22] gedacht. Eine Bewertung der verschiedenen Mikroprozessoren hinsichtlich eventueller Anwendungsfälle wurde von HABGOOD [8] veröffentlicht.

Wie bei jedem anderen Rechnersystem muß der Mikrorechner-Anwender, bevor er die Eigenschaften des Systems optimal nutzen kann, mit einem gewissen Anwendungs-know-how vertraut sein. Oft wird jedoch auf diesem Gebiet Neuland betreten, so daß eine relativ große Einarbeitungszeit erforderlich ist [2]. Besonders die z.Zt. geringe Softwareunterstützung erschwert die Erstellung der erforderlichen Programme [40]. So erfolgt die Programmierung von Mikrorechnern im allgemeinen in einer Assemblersprache. Die Übersetzung und der Test der erstellten Programme wird - wie an anderer Stelle noch

eingehend dargestellt - anhand eines sogenannten 'Mikrorechner-Entwicklungssystems' oder mittels 'Cross-Software' auf einer größeren Rechenanlage durchgeführt.

1.2 Zielsetzung

Um den Anwender möglichst weitgehend von rechnerspezifischen Kenntnissen zu entlasten und ihm eine weniger fehlerintensive, selbstdokumentierende Programmierung von Mikrorechnern zu ermöglichen, kann auf die Verwendung einer höheren Programmiersprache nicht verzichtet werden [34]. Alle bisher bekannten Konzepte zur Programmierung von Mikrorechnern in einer höheren Programmiersprache setzen entweder ein relativ umfangreiches Entwicklungssystem mit Floppy Disk [35] oder eine größere Rechenanlage zur 'Cross-Compilation' voraus. Dies bedeutet, daß die Programmerstellung an bestimmte Örtlichkeiten gebunden ist und nur in Einzelfällen am Prozeß selbst abgewickelt werden kann, was sich u.a. besonders in der Inbetriebnahmephase störend auswirkt. Zusätzlich weisen die eingesetzten Compiler (PL/M bzw. PL/M Plus) nur wenige Prozeßsprachelemente auf.

Eine befriedigende, mit möglichst geringem Zeitaufwand verbundene Programmerstellung, kann besonders durch eine interaktive Arbeitsweise [57] erreicht werden. Sieht man von entsprechender Cross-Software für größere Rechenanlagen ab, so ist bis heute kein Konzept dieser Art für Mikrorechner bekannt.

Um ein vorhandenes Mikrorechnersystem unabhängig von anderen Systemen programmieren zu können, muß - wie auch bereits von DOLL [3] gefordert - ein sogenannter 'Inselbetrieb' möglich sein. Außer einer Standard-Fernschreibmaschine mit Anbaulochleser und -stanzer und den entsprechenden Ein/Ausgaberroutinen, sollten an den Systemausbau und die vorhandene Software keine weiteren Forderungen gestellt werden.

Das Ziel der Arbeit besteht nun darin, die oben angeführten Lücken zu schließen, also ein Konzept zu erarbeiten, daß eine interaktive Programmierung von Mikrorechnern im Inselbetrieb ohne Störung des jeweiligen Prozeßablaufes zuläßt.

2. MÖGLICHKEITEN DER SOFTWAREERSTELLUNG BEI MIKRORECHNERN

Während im Jahre 1971 der erste funktionsfähige Mikroprozessor vorgestellt wurde, stehen z.Zt. etwa 25 verschiedene Typen zur Verfügung oder sind zumindest angekündigt. Bei dieser Vielzahl der Systeme stellt der Prozessor *INTEL 8080* eine 'Quasi-Norm' dar [19], da bis zu diesem Zeitpunkt keine offizielle Normung für Mikroprozessoren einzuführen war. Da außerdem die derzeitige Vorgehensweise bei der Softwareerstellung für die verschiedenen Mikrorechner praktisch identisch ist, kann im folgenden zur einführenden Beschreibung der Hard- und Softwareeigenschaften von Mikrorechnern das System *INTEL 8080* betrachtet werden.

2.1 Eigenschaften der Mikrorechnersysteme

Eine detaillierte Übersicht über die speziellen Mikrorechnereigenschaften und die verschiedenen Prozessortypen ist den Aufsätzen von OGDIN [19], SAWIN [21] sowie HOLT/LEMAS [10] zu entnehmen. In diesem Kapitel soll nur dann auf spezielle Systemeigenschaften eingegangen werden, sofern dies zur Verdeutlichung der derzeitigen Vorgehensweise bei der Softwareerstellung vorteilhaft erscheint.

Definition 2.1

Der *Mikroprozessor* ist die Zentraleinheit eines Kleinstrechners in hochintegrierter Halbleitertechnologie, wobei sich die gesamte CPU mit allen Registern sowie Befehls- und Rechenwerk auf sehr wenigen - oft nur einem - Chip befindet.

Es ist zu beachten, daß der Begriff des Mikroprozessors wie er in diesem Beitrag nach Def. 2.1 verwendet wird, keine Aussage über die Mikroprogrammierung des Systems enthält und somit nur die Kleinheit der Baugruppe charakterisiert.

Die prinzipielle Hardwarestruktur eines Mikroprozessors ist aus Abb. 2 ersichtlich. Mikroprozessoren sind 'bus orientiert'. Neben der ALU, dem Befehlsdekoder und dem Programmzähler ist im wesentlichen noch eine

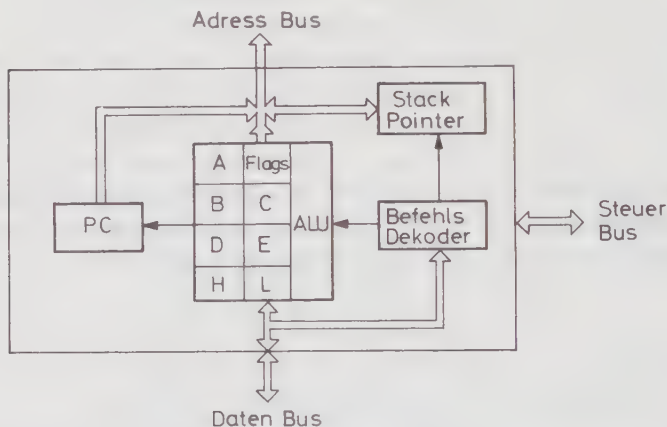


Abb. 2: Prinzipielle Hardwarestruktur eines Mikroprozessors
Beispiel: INTEL 8080

Anzahl von Registern integriert. Der erforderliche Arbeitsstack befindet sich meist außerhalb der CPU in dem vorhandenen RAM-Bereich, während der zugehörige Stackpointer dann wiederum Bestandteil des Prozessors ist.

Ebenso wie bei größeren Rechnersystemen, kennt man auch bei Mikroprozessoren verschiedene Adressierungsmöglichkeiten. Einen Überblick hierzu findet man bei OGDIN [19]. Eine für Mikroprozessoren besonders günstige und oft angewandte Adressierungsmöglichkeit stellt die Methode der *register indirect addressing* dar. Dabei wird der Inhalt eines oder mehrerer Register zur Auswahl eines Speicherplatzes benützt. Da der von Mikroprozessoren zu adressierende Speicherbereich meistens bis zu 64k Wörter beträgt, sind für eine derartige Adressierungsform bei dem bytestrukturierten 'Quasi-Norm' System INTEL 8080 genau zwei Register erforderlich. Diese Tatsache wird von dem Befehlsvorrat dahingehend unterstützt, daß sich bestimmte Registerpaare - wie auch in Abb. 2 angedeutet wurde - mit einem Befehl ansprechen lassen (Registerpaare BC, DE, HL). Bei dem betrachteten System kann keine relative Adressierung durchgeführt werden. Deshalb muß der Objektcode in absolut adressierter Form vorliegen, was besonders - wie sich später zeigen wird - die Arbeit des zu konzipierenden *linkage editors* erschwert.

Die nun folgenden Definitionen sollen zu einer weiteren Begriffsklärung beitragen.

Definition 2.2

Der Begriff *Mikrorechner* kennzeichnet einen funktionsfähigen Kleinstrechner, der neben dem eigentlichen Rechelement - dem Mikroprozessor - aus weiteren zur Funktion des Gesamtsystems erforderlichen Hardwarekomponenten wie Speichermodule, Interfaces usw. besteht.

Definition 2.3

Ein *Mikrorechner-Entwicklungssystem* ist ein Mikrorechner, bei dem aufgrund zusätzlicher Hard- und Software eine vollständige Programmerstellung, der Test der Programme und die Speicherung des ausgetesteten Objektprogramms in einem nicht flüchtigen Speichermedium (z.B. in PROM's) möglich ist.

Nach der oben durchgeführten Begriffsklärung soll nachfolgend in zusammengefaßter Form die derzeitige Vorgehensweise bei der Softwareerstellung für Mikrorechner dargestellt werden.

2.2 Softwareunterstützung bei der Erstellung von Programmen

Mikrorechner lassen sich auf drei verschiedenen Wegen programmieren und zwar

- direkt in Maschinensprache,
- in einer Assemblersprache oder
- in einer höheren, anwenderorientierten Sprache.

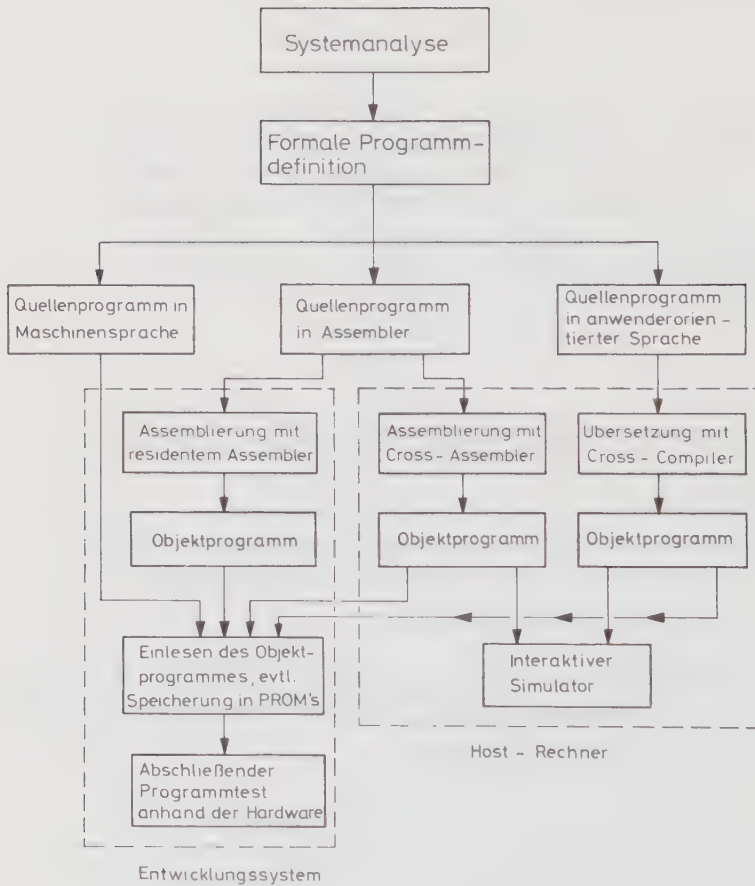


Abb.3: Vorgehensweise bei der Softwareerstellung für Mikrorechner

Die direkte Programmierung in Maschinencode sollte nur als Notlösung und bei sehr kleinen Programmen in Betracht gezogen werden, zumal in diesem Fall von dem Anwender eine bestimmte Hardwarekenntnis gefordert wird. Die derzeitigen Möglichkeiten Mikrorechner effektiv zu programmieren zeigt Abb. 3.

Eine erste Unterstützung bei der Programmerstellung stellt dabei das zuvor erwähnte Mikrorechner Entwicklungssystem dar. Ein derartiges Entwicklungssystem wird vom Hersteller neben der erforderlichen Hardware - wie z.B. Front Panel, I/O-Interfaces, Mindestausbau an Speicherplatz, Programmier-einrichtung für PROM's - mit folgenden Softwarekomponenten geliefert:

- Testmonitor für den Teletype-Verkehr und Hardwaretest erstellter Programme,
- Editor-Programm zur einfachen Bearbeitung des Quellprogramms,
- Assembler-Programm zur Assemblierung von Quellprogrammen.

Auf der linken Seite von Abb. 3 wurde die Vorgehensweise bei der Assemblierung bei Einsatz des Entwicklungssystems skizziert. Eine derartige Vorgehensweise erweist sich jedoch als sehr zeitaufwendig [33]. So muß im einzelnen

- der Assembler eingelesen,
- der Lochstreifen mit dem Quellprogramm mehrfach eingelesen (da Mehr-Pass-Assembler),
- ein Lochstreifen mit dem Objektprogramm ausgegeben und
- das erzeugte Objektprogramm eingelesen werden.

Sollte das Quellprogramm noch fehlerbehaftet sein, so ist eine Bearbeitung mit dem Text-Editor erforderlich, bevor eine erneute Assemblierung erfolgt. Die Programmerstellung schließt mit dem erfolgreichen Hardwaretest ab.

Zukünftig sollen Entwicklungssysteme für Mikroprozessoren mit einem Plattensystem (Floppy Disk) ausgestattet sein, welches von einem DOS (Disk Operating System) verwaltet wird [16]. Durch diese Möglichkeit kann das einmal eingegebene Quellprogramm sofort bearbeitet werden, was zu einer wesent-

lichen Erleichterung bei der Programmerstellung führt.

Hat ein Benutzer Zugang zu einer größeren Rechananlage, so wird der Programmierungsvorgang durch die Verwendung sogenannter Cross-Software [33] erleichtert. Es stehen verschiedene Cross-Produkte zur Verfügung, je nachdem ob das Quellprogramm in einer Assemblersprache oder in einer höheren Programmiersprache geschrieben wurde. Durch die Möglichkeit der Programmierung in einer höheren Programmiersprache [13, 16], können u.a. auch Anwender, die weniger mit der Hardwarestruktur des Mikroprozessors vertraut sind, eine Programmierung dieser Systeme durchführen. Mit einem zusätzlichen Simulator des entsprechenden Mikroprozessors ist ein weitgehender, interaktiver Test der erstellten Programme auf dem Host-Rechner (Abb.3, rechts) möglich.

2.3 Grenzen der derzeitigen Programmierungsverfahren bei Mikrorechnern

Wie bereits vorher erwähnt, muß gegenwärtig der größte Teil der Software für Mikroprozessoren in einer Assemblersprache geschrieben werden, da dem Programmierer nur ein Entwicklungssystem zur Verfügung steht, für das u.a. nach der Meinung verschiedener Fachleute bei dem derzeitigen Entwicklungsstand kein Compiler zu implementieren ist [31, 33], oder die entsprechende Cross-Software bei dem ausgewählten Mikroprozessor noch nicht ausgeliefert war.

Von SCHNEIDER [40] wurde für die fehlerfreie Programmierung von Mikrorechnern in einer Assemblersprache ein Erfahrungswert von 1-15 Befehlen je Stunde angegeben, wobei Programmierer mit durchschnittlichen Kenntnissen der Softwareerstellung etwa 3 Befehle pro Stunde erzeugen. Diese Angaben entsprechen ungefähr den Werten, die aufgrund der eigenen Tätigkeit ermittelt wurden. Es ist deshalb abzusehen, daß eine derartige Vorgehensweise sehr zeitintensiv ist. So erfordert die Erstellung eines Programmes von ungefähr 4k Byte Länge einen Zeitraum von etwa 7 Mann-Monaten. Schließlich erschwert zusätzlich mangelhafte Firmensoftware eine zügige Programmentwicklung. Auch ist zu beachten, daß bei einer Programmierung in Assemblersprache in jeder Erstellungsphase gleichzeitig eine ausführliche Dokumentation der bisher geleisteten Arbeit er-

folgen muß, um Kommunikationsschwierigkeiten zwischen den einzelnen Mitarbeitern des Projektes auszuschalten.

Bezeichnet man bei der Realisierung einer gestellten Aufgabe mit

t_g = erforderliche Gesamtzeit,

t_A = Zeit für die Analyse eines Problems und Entwicklung
eines Konzeptes,

t_I = Eigentliche Implementierungszeit,

t_T = Zeit für Test und Korrektur des erstellten Programmes,

so kann nach [40] für den Einfluß der einzelnen Phasen auf die Softwarekosten die Gleichung

$$t_g = t_A + t_I + t_T$$

$$(1) \quad t_A : t_I : t_T = 2 : 1 : 2$$

angegeben werden. Während nun t_A von der eigentlichen Programmierungsstrategie unbeeinflussbar ist, sind die Zeiten t_I und t_T , die nach Gleichung (1) zusammen 60 % der Gesamtzeit beanspruchen, durchaus durch die gewählte Vorgehensweise bei der Programmierung zu beeinflussen.

Ein wichtiger Punkt zur Senkung der Softwarekosten stellt dabei die Programmierung in einer höheren Programmiersprache dar [19, 31, 33, 34, 40] . So sind in erster Linie zu nennen:

- Selbstdokumentierung der Programme,
- Austauschbarkeit von Softwaremodulen [39],
- Systemsicherheit [44].

Die Firma INTEL wurde als erste dieser Forderung nach einer höheren Programmiersprache durch Erstellung eines Cross-Compilers für die Programmiersprache PL/M, einer Abwandlung der bereits eingeführten Sprache PL 1, gerecht [29, 31] . Es zeigt sich jedoch in der Praxis, daß bei Programmen kleineren Umfangs (etwa bis 1k Byte [40]) die in dieser Sprache erstellt werden, mindestens 50 % mehr Speicherplatz für den Objektcode erforderlich ist, als bei entsprechenden Assemblerquellen.

Bei größeren Programmen dagegen ist dies nicht mehr der Fall, da dann die Struktur des erzeugten Objektcodes von dem Compiler effektiver angelegt wird als von einem durchschnittlichen Assembler-Programmierer durchzuführen wäre [31]. Berücksichtigt man schließlich, daß die Kosten für den erforderlichen Speicherplatz immer mehr sinken (Ende 1975 kostet eine 16k-8 Bit RAM-Platine etwas mehr als 3000 DM), so sind die Speicherplatzkosten für den etwas längeren Objektcode im Hinblick auf die einfache Programmierung vernachlässigbar.

Die Erfahrung zeigt, daß sich die Handhabung der Cross-Software zur Compilation von Quellenprogrammen und zum interaktiven Test des erzeugten Objektcodes relativ unhandlich gestaltet, da die jeweiligen Großrechnersysteme durch anderweitigen Teilnehmerbetrieb meist überlastet sind, so daß notwendige Übersetzungen aus Rechenzeitgründen oft nur nachts durchgeführt werden können. Die interaktive Arbeitsweise ist jedoch ein wesentlicher Punkt für eine optimale kostengünstige Programmerstellung. Zur Umgehung der oben angedeuteten Begrenzungen bei der derzeitigen Softwareerstellung für Mikroprozessoren ist deshalb die Realisierung der Aufgabe:

- *interaktive Programmierung von Mikrorechnern in einer höheren Programmiersprache bei dem jeweils vorhandenen Mikrorechner* - erforderlich. In den folgenden Kapiteln wird ein Konzept dargestellt, welches eine interaktive Programmierung von Mikrorechnern im Inselbetrieb, d.h. unabhängig von einer größeren Rechanlage, ohne Störung eines evtl. von diesem System bereits kontrollierten Prozesses erlaubt.

3. EIN ÜBERSETZERKONZEPT FÜR MIKRORECHNER

Die folgenden Betrachtungen sollen zur Einführung in das Konzept eines Übersetzers für Mikrorechner dienen. Es wird die prinzipielle Struktur des Übersetzers mit den jeweiligen Systemkomponenten vorgestellt. Eine exakte Darstellung des Systemkonzeptes ist aus den sich anschließenden Kapiteln ersichtlich.

3.1 Forderungen an einen anwenderfreundlichen Übersetzer für Mikrorechner

3.1.1 *Benutzerfreundlichkeit*

Um die Softwarekosten zu verringern, ist eine möglichst umfassende Unterstützung des Benutzers bei der Softwareerstellung erforderlich. Dabei muß in erster Linie der Anwender seine Programme zur Lösung einer vorgegebenen Aufgabe selbst erstellen können, ohne umfangreiche Programmierkenntnisse zu besitzen [58]. Dies erfordert jedoch, daß

- eine höhere Programmiersprache mit einfacher Syntax zur Verfügung steht, die bereits weit verbreitet und somit einem großen Anwenderkreis bekannt ist,
- eine möglichst umfassende zeilenweise Analyse des eingegebenen Quellenprogrammes durch einen interaktiven Übersetzer möglich ist,
- die Programmerstellung interaktiv erfolgen kann,
- eine ausreichende Softwareunterstützung für den Programmierer (z.B. TRACE-Eigenschaft) gewährt wird und
- das Übersetzersystem von dem Benutzer einfach zu bedienen ist.

3.1.2 *Prozeßanforderung*

Außer den oben genannten Punkten sind zusätzlich noch Forderungen des zu programmierenden Prozesses an das Übersetzersystem zu berücksichtigen.

So ist u.a. zu nennen:

- Die Programmiersprache sollte die Gegebenheiten in der Prozeß-rechentechnik berücksichtigen,
- das Übersetzersystem sollte nicht nur bei der Neuerstellung von Programmen, sondern auch bei Programmänderungen und Programmtest, z.B. während der Inbetriebnahmephase den Anwender unterstützen,
- während der Zeitspanne einer erforderlichen Programmänderung darf der Prozeßablauf selbst nicht unterbrochen werden.

3.1.3 Hardwarebegrenzungen

Da die gestellte Aufgabe eine Programmierung von Mikrorechnern zum Ziele hat, sollte dies auch bei Systemen möglich sein, die sich durch einen minimalen Ausbau von Speicherplatz und Peripherie auszeichnen. Dies bedeutet, daß

- der Übersetzer den zu programmierenden Mikrorechner softwaremäßig nicht belasten sollte,
- das Übersetzersystem im *Inselbetrieb* arbeiten muß, d.h. nach DOLL [3], ohne zusätzliche größere DV-Anlage im Sinne einer Programm- und Datenträgeraufbereitung auskommt und
- zur Programmierung des entsprechenden Mikrorechners als einziges peripheres Gerät nur eine Standard-Fernschreibmaschine (TTY) zur Verfügung steht.

Im weiteren ist zu beachten, daß der Programmiervorgang möglichst weitgehend systemunabhängig abgewickelt werden kann, um eine kostengünstige Softwareerstellung zu ermöglichen.

3.2 Konzept eines Übersetzers für Mikrorechner

Jedes Rechnersystem erfordert zur Kommunikation mit seiner Umgebung, je nach Leistungsfähigkeit und Anwendungsfall, eine entsprechende Anzahl peripherer Geräte. Beschränkt man sich im einfachsten Fall auf eine Standard-Fernschreib-

maschine (TTY), so stellt diese Möglichkeit zum heutigen Zeitpunkt noch die wirtschaftlichste Lösung zum Informationsaustausch zwischen einem kleineren Rechnersystem und der jeweiligen Umgebung dar [3]. Bei Mikrorechnern ist

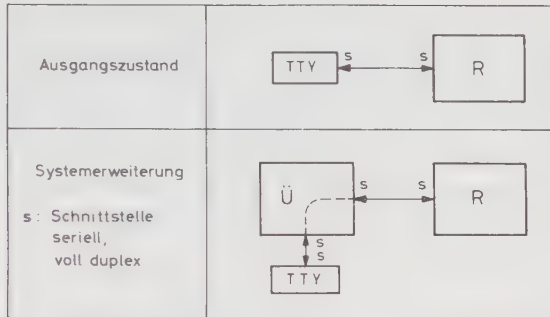


Abb. 4: Prinzip der Ankopplung des Übersetzungssystems U an den Rechner R.

diese nach Abb. 4 oben gezeigte Systemkonfiguration allgemein üblich oder zumindest ohne besonderen Aufwand realisierbar. Deshalb kann für die Programmierung von Mikrorechnern folgende Voraussetzung gemacht werden:

Definition 3.1

Der zu programmierende Mikrorechner ist als Mindestperipherie mit einer Standard-Fernschreibmaschine (TTY) ausgerüstet. Der TTY-Verkehr wird von bereits implementierten I/O-Routinen abgewickelt.

Gemäß den in Kapitel 3.1 aufgestellten Forderungen soll das benötigte Übersetzersystem U den zu programmierenden Rechner R nicht belasten, um auch Rechner mit minimaler Systemkonfiguration programmieren zu können. Die in Abb. 4 vorgestellte Systemerweiterung um das nachfolgend zu konkretisierende Übersetzersystem U erweist sich dabei als eine einfach zu realisierende Möglichkeit ohne Schnittstellenprobleme, da das Übersetzersystem U schnell in die voraussetzungsgemäß vorhandene Systemkonfiguration anhand der jeweiligen seriellen Schnittstellen zu integrieren ist.

Definition 3.2

Das Übersetzersystem Ü ist ein aktives System mit zwei seriellen Schnittstellen zum Datenaustausch mit dem zu programmierenden Mikrorechner und der vorhandenen Ein-/Ausgabereinheit.

Durch diese Vorgehensweise ist u.a. eine Programmerstellung ohne Störung des jeweiligen Prozeßablaufes bei eventuell gleichzeitiger Prozeßprotokollierung usw. möglich. Zur Realisierung der vielfältigen Anforderungen an das Übersetzersystem wurden neueste Erkenntnisse für die interaktive Programmerstellung berücksichtigt und teilweise neue Konzepte vorgeschlagen und implementiert. In dem Übersetzersystem Ü sind im wesentlichen die in Abb. 5 gezeigten Module integriert, wobei entsprechend Definition 3.1 kein zusätzlicher

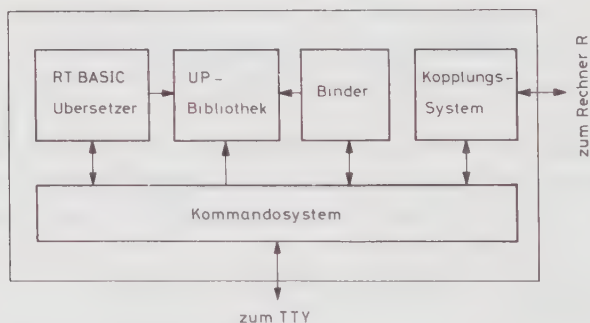


Abb. 5: Prinzipielle Struktur des Übersetzersystems Ü.

Hintergrundspeicher zur Verfügung steht. Die Vorstellung und die Diskussion der jeweiligen Module des Übersetzersystemkonzeptes, d.h.

- des dialogfähigen Real-Time BASIC-Übersetzers,
- der Unterprogramm-bibliothek für den Mikroprozessor *INTEL 8080*,
- des Binders für absolut adressierten Objektcode,
- des Kopplungssystems zur Verwaltung des Datenaustausches zwischen R und Ü und

- des Kommandosystems zur Unterstützung des Benutzers bei der Programmerstellung, Einfügen und Binden von Programmteilen, dem Programmtest und vielen weiteren Aufgaben,

wird in den folgenden Kapiteln behandelt. Entgegen der bisherigen Vorgehensweise bei der Softwareerstellung für Mikrorechner nach Abb. 3, ist bei Einsatz des obengenannten Übersetzersystems eine interaktive Programmerstellung im Inselbetrieb ohne Störung eines eventuellen Prozeßablaufes möglich, wie es zusammenfassend in Abb. 6 angedeutet wurde.

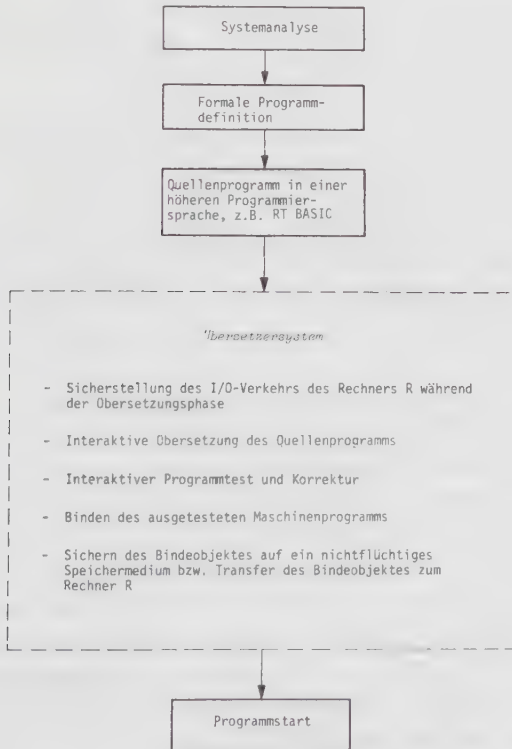


Abb. 6: Anzustrebende Vorgehensweise bei der Softwareerstellung für Mikrorechner

4. AUSWAHL EINER GEEIGNETEN PROGRAMMIERSPRACHE

4.1 Betrachtung bekannter Programmiersprachen

Erfahrungsgemäß sind Rechnerausfälle infolge von Fehlern in der Anwendersoftware am häufigsten [44]. Um bei der Programmerstellung Fehler möglichst zu vermeiden, sollte die Programmierung - wie bereits schon mehrmals angesprochen - weitgehend in einer höheren Programmiersprache vom jeweiligen Anwender selbst durchgeführt werden. Deshalb muß die zu verwendende Programmiersprache

- leicht zu erlernen sein,
- einprägsam und einem möglichst großen Anwenderkreis bereits bekannt sein,
- sich durch eine einfache Syntax auszeichnen,
- Sprachelemente sowohl für den Bereich der numerischen und einfachen nichtnumerischen Datenverarbeitung, als auch für die Prozeßautomatisierung besitzen,
- für häufig wechselnde Echtzeitanwendungen geeignet sein,
- die Möglichkeit der unmittelbaren Syntaxerkennung zulassen,
- interaktive Programmerstellung und Test gestatten und
- vom Umfang her so beschaffen sein, daß sie bei einem Mikrorechnersystem implementiert werden kann [48,54,58].

Es ist einzusehen, daß nicht alle der obengenannten Forderungen - die zudem teilweise widersprüchlich sind - sich in einer einzigen Sprachkonstruktion optimal realisieren lassen, d.h. es müssen Kompromisse geschlossen werden. Die Programmierung von Prozeßrechnern sollte in einer Prozeßsprache erfolgen, die eine Erweiterung einer bereits eingeführten Sprache darstellt. Nach FRAADE [46] stehen dabei u.a. zur Diskussion:

BASIC, CORAL 66, FORTRAN, PEARL, PROCAL und RTL/2.

Auch PL/1 sollte in diesem Zusammenhang nicht unerwähnt bleiben. PEARL befindet sich z.Zt. noch im Entwicklungsstadium und wird bei Kleinstsystemen wohl kaum zu implementieren sein [64]. Die Sprachen CORAL 66 und RTL/2 erfüllen z.B. nicht die Forderung, daß Ein-/Ausgabevorgänge, Prozeßstarts usw. ausschließlich mit Mitteln der Sprache durchgeführt werden können und einen besonderen Betriebssystemaufruf erfordern [48,49]. Nach MARTINES [52] wäre aus Standardisierungsgründen bei Erweiterungen eine Konzentration auf die Sprachen BASIC, FORTRAN und PL/1 wünschenswert. Diese Forderung wird in [46] nochmals eingeeengt auf BASIC und FORTRAN.

Verschiedene Mikroprozessorhersteller ermöglichen die Programmierung ihrer Systeme in der höheren Programmiersprache PL/M, einer Untermenge von PL/1, unter Einsatz entsprechender Cross-Software [29]. Diese Sprache ist blockstrukturiert und erschwert somit eine einfache interaktive Arbeitsweise.

BASIC zeichnet sich gegenüber FORTRAN durch einfachere grammatische Regeln aus, ist in kürzester Zeit erlernbar, besitzt keine Blockstruktur und läßt somit die Möglichkeit der interaktiven Programmierung auf einfache Art und Weise zu. Außerdem wird BASIC bereits von einem großen Anwenderkreis eingesetzt. Aus diesen Gründen hat sich BASIC als *Quasi-Standard* der meisten Minicomputersysteme entwickelt wie aus der Prozeßrechnerübersicht von PAUL/SCHULER [55] ersichtlich ist.

Der Einsatzbereich von Mikrorechnern wird durch die zu bewältigenden Aufgaben begrenzt. Zur Abwicklung komplexerer Aufgaben stehen die entsprechend größeren Rechnersysteme zur Verfügung. Deshalb stellt die Programmiersprache BASIC - eine Sprache, die ebenfalls zur interaktiven Programmierung kleinerer Probleme entwickelt wurde - eine günstige Basis zur Programmierung von Mikrorechnern dar.

4.2 Real-Time BASIC

Erweitert man den Sprachvorrat der Programmiersprache BASIC im Hinblick auf Echtzeitanwendungen und Prozeß-Ein-/Ausgabevorgänge so wird die dann vorliegende Spracherweiterung oft durch den Begriff *Real-Time BASIC* (RT BASIC) gekennzeichnet. Es sind u.a. folgende BASIC-Erweiterungen für Prozeßanwendungen

bekannt:

- ADAPTS (Varian Extended BASIC)
- BASEX (Phys. Institut der Universität Freiburg)
- BASIC/RT 11 (DEC)
- FPL (Foxboro)
- INDUSTRIAL BASIC (DEC)
- PROCESS BASIC (Unicomp)
- RTE-B (Hewlett Packard)
- KENT K 90 BASIC
- BASIC B (Schlumberger)

Die Semantik dieser Spracherweiterungen wird in dem *First Report of the International Purdue Workshop* [58] gegenübergestellt und diskutiert. Jeder dieser *BASIC-Dialekte* zeichnet sich dabei durch ganz bestimmte Eigenschaften aus. So können z.B. bei PROCESS BASIC Assembler-Anweisungen direkt in ein BASIC-Programm zeilenweise eingefügt werden, während bei anderen Dialekten Assembler- bzw. Objektcodeprogramme mit dem Schlüsselwort CALL anzuschließen sind. Es ließe sich auf diese Art und Weise noch eine Vielzahl derartiger Unterschiede aufzeigen, die oft auch durch die jeweilige Hardware bestimmt werden.

Besonders durch Einführung der in der Syntax den einfach indizierten Variablen entsprechenden Prozeßvariablen, welche die Verbindung zu dem äußeren Environment (Prozeßgeschehen) herstellen und den einfachen Echtzeitsprachelementen, erscheint die Prozeßsprache BASEX als eine sinnvolle Spracherweiterung für die Implementierung bei Mikrorechnern. Eine genaue Darstellung der Syntax dieser Prozeßsprache findet man in dem Bericht C 25 des Instituts für Physik der Universität Freiburg [47]. An dieser Stelle sollen nur die wesentlichen Erweiterungen kurz genannt werden. So sind u.a. folgende Abweichungen gegenüber Dartmouth-BASIC zu vermerken:

- Namen für Variable können bis zu 4 Zeichen lang sein,
- Möglichkeit der Verknüpfungen von Strings durch Stringoperatoren,

- Einführung eines CALL-Statements zum Anschluß von Objektcodeprogrammen,
- Einführung von Prozeßvariablen und
- zusätzliche Sprachmittel zur Behandlung von Unterbrechungen und Zeitbedingungen.

Bei der Konzeption des Übersetzers steht die Benutzerfreundlichkeit im Vordergrund. Es ist deshalb besonders wichtig, daß die Programmierung eines Systems nicht durch komplizierte Sprachkonstruktionen behindert wird. Dies ist besonders im Fall der Prozeßsprache BASEX, z.B. durch Einsatz der Prozeßvariablen, weitgehend ausgeschaltet. Da zusätzlich diese Sprachkonstruktion in seinem Basissprachumfang am wenigsten systembezogen erscheint und für Kleinstrechner konzipiert ist, wurde BASEX als Grundlage der Implementierung einer höheren Programmiersprache für Mikrorechner ausgewählt.

4.3 BASEX für Mikrorechner: BASEM

Dieses Kapitel dient der Diskussion durchzuführender Änderungen bezüglich des BASEX Sprachumfanges für die Implementierung bei Mikrorechnern¹. Es werden Korrekturen hinsichtlich der Syntax und Semantik begründet und eine genaue Beschreibung der Syntax in EBNF angegeben. Zur Kennzeichnung dieser modifizierten Sprachkonstruktion "BASEX FOR MICROCOMPUTERS", wird die gekürzte Bezeichnung "BASEM" eingeführt.

4.3.1 Konzept durchzuführender Modifikationen

Voraussetzungsgemäß sollte die Softwareerstellung für Mikrorechner in einer höheren Programmiersprache ohne zusätzlichen Hintergrundspeicher erfolgen. Die BASEX-Statements zur Segmentierung von Benutzerprogrammen werden somit nicht benötigt, da sie nur bei Systemen mit Hintergrundspeicher sinnvoll erscheinen [47]. Ebenso verhält es sich mit Kommandos, die Hintergrundspeicher bezogen sind, wie z.B. SAVE oder LOAD. Bei BASEM wurde deshalb auf eine Übernahme derartiger Sprachelemente verzichtet.

¹ Aus Standardisierungsgründen wurde auf die Einführung einer vollständig neuen Sprachkonstruktion verzichtet

String-Expressions

Ebenso wie BASEX kennt auch BASEM keinen Datentyp `<integer>`. Numerische und logische Größen sind stets vom Typ `<real>`. Zusätzlich ist nur der Datentyp `<string>` erlaubt.

Die softwaremäßige Abwicklung arithmetischer Verknüpfungen erfordert nach [26] bei dem betrachteten Mikrorechnersystem für Gleitkommaoperationen einen oft wesentlich höheren Rechenzeitbedarf gegenüber der entsprechenden Festkommaoperation. Da der Arbeitsgeschwindigkeit von Mikroprozessoren auf Grund der angewandten Technologie eine relativ niedrige Grenze gesetzt ist (der Grenzwert für die Taktfrequenz des betrachteten Mikroprozessors INTEL 8080 beträgt z.B. etwa 2 MHz), sollte zur Verbesserung der allgemeinen Programmlaufzeit der Programmierer im Bedarfsfall auf die Möglichkeit der Festkommaoperation zurückgreifen können.

Vergleicht man unter dem oben genannten Aspekt Unterprogramme für Fest-Gleitkommaoperationen hinsichtlich Rechenzeit und erforderlichem Speicherplatz am Beispiel der Addition zweier ganzzahliger Operanden, so läßt sich folgendes feststellen:

Vorausgesetzt, daß die erforderlichen Randbedingungen zur fehlerfreien Durchführung der Operation erfüllt sind, ist nach [26] bei Durchführung der Addition unter Verwendung der Gleitkommarechnung, die etwa siebenfache Rechenzeit und der vierfache Speicherplatz für das entsprechende Unterprogramm im Gegensatz zur Festkommarechnung notwendig.

Natürlich wird dieses Mißverhältnis bezüglich der Rechenzeit bei Verwendung von Gleitkommarechenwerken nicht mehr gegeben sein. Es wäre aber unrealistisch vorauszusetzen, daß jedes kleine Mikrorechner-System, welches der Benutzer im Labor selbst entwickelt, ein derartiges Rechenwerk besitzt. Aus den oben angedeuteten Gründen wurde für BASEM die folgende Konzeption erstellt, wobei die jeweils exakten semantischen oder syntaktischen Definitionen aus den Kapiteln 4.3.2 und 4.3.3 zu entnehmen sind.

Definition 4.1

BASEM kennt wie BASEX nur die Datentypen `<real>` und `<string>`

Definition 4.2

Die Länge von Stringvariablen beträgt - sofern sie nicht gesondert über die CHAR-Anweisung festgelegt wird - zwei Zeichen (Bytes). Dies entspricht der Verarbeitungsbreite der Festkommaunterprogramme.

Bei BASEM wurde der Begriff des Ausdruckes nahezu ohne Einschränkung auch auf Strings erweitert. Dieser Begriff erscheint bereits in den verschiedenen Standardisierungsvorschlägen internationaler Komitees [72, 87], steht jedoch dort nur für Stringkonstanten oder Stringvariable. Neben der syntaktisch korrekten Zusammensetzung eines Ausdruckes mit Satzelementen vom Typ `<real>`, ist nun eine ähnliche Form mit Satzelementen vom Typ `<string>` zulässig, d.h.

$$\langle \text{expr.} \rangle :: = \langle \text{real-expr.} \rangle \mid \langle \text{string-expr.} \rangle$$

Definition 4.3

Die Wirkungsweise von Operatoren bei `<string-expr.>` ist - logische Operatoren ausgenommen - ausschließlich auf die ersten zwei Zeichen (Bytes) der Strings beschränkt.

Die "Philosophie" zur Einführung obiger Definitionen sei nachfolgend kurz erläutert.

Mit BASIC steht eine Programmiersprache zur Verfügung, die u.a. auch auf Grund der wenigen Datentypen `<real>` und `<string>`, die zudem nicht explizit deklariert werden müssen, auch für den Anfänger ein schnelles Einarbeiten gestattet. Wie bereits erwähnt, wurde auch bei BASEX kein Datentyp `<integer>` eingeführt, so daß alle numerischen und logischen Größen vom Typ `<real>` sind. Dem Nachteil der größeren Rechenzeit steht der Vorteil einer unproblematischen Programmerstellung gegenüber.

Einem mit der Programmiersprache und dem System vertrauteren Benutzer sollte jedoch die Möglichkeit offen stehen, den Vorteil der geringeren Rechenzeit von Festkommaoperationen auszuschöpfen. Für diesen Zweck läßt sich die Einführung eines Datentypes `<integer>` jedoch nur dann vermeiden, wenn arithmetische Operationen mit Strings zulässig sind. Es werden keine zusätzlichen Deklarationen benötigt, so daß sich für den mit der Program-

miersprache BASIC weniger vertrauten Benutzer gegenüber seiner bisherigen Programmerstellung keine Änderungen ergeben. Er wird von dieser Möglichkeit der *String-Expressions* keinen Gebrauch machen. Dieses Verfahren soll jedoch den erfahreneren Anwender bei der Programmierung der relativ "langsamen" Mikroprozessoren in einer höheren Programmiersprache unterstützen.

Die etwas ungewohnte Schreibweise der *String-Expressions* durch die Verknüpfung der Strings mit arithmetischen Operatoren, z.B. in der folgenden Anweisung

$$10 \text{ LET } A\beta = B\beta * (C\beta - D\beta),$$

wird der Anwender vermutlich nicht als störend empfinden, da es sich bis auf die andere Art der Variablen um die vertraute Form eines Ausdruckes handelt. Jedoch nicht nur bei LET-Anweisungen sollte der erfahrenerer Programmierer mit *String-Expressions* arbeiten, sondern z.B. auch zur Steigerung der Rechengeschwindigkeit bei FOR-Schleifen, bei der Handhabung spezieller Prozeßobjekte, zur Sortierung von Strings über IF-Abfragen und allen weiteren Anwendungsfällen, die eine möglichst geringe Rechenzeit erfordern.

Prozeßobjekte

Neben der Möglichkeit spezielle, zur Abwicklung von Prozeßaufgaben vom Benutzer in Assembler geschriebene Programme auf der Ebene der höheren Programmiersprache durch das Statement "CALL" aufzurufen, wurden in BASEX noch die sogenannten Prozeßvariablen eingeführt. Es handelt sich dabei um einen Variablentyp, der in ausführbaren Anweisungen syntaktisch wie eine gewöhnliche, allerdings höchstens einfach indizierte Variable zu behandeln ist und bei jeder Benutzung einen definierten Ein- oder Ausgabeprozess einleitet [48].

Die einfache Handhabung dieser Prozeßvariablen, für die nur ein einziger Parameter, nämlich die "Kanalnummer" des jeweiligen Prozeßinterfaces zulässig ist, spricht für ihre Implementierung auch bei Mikrorechnern. Deshalb wurde auch in BASEM diese Möglichkeit des Datenaustausches zwischen Prozeßgeschehen und Rechensystem vorgesehen. Folgende Modifikationen der

entsprechenden BASEX Semantik und Syntax erschienen jedoch für die Implementierung bei Kleinstrechnersystemen zweckmäßig:

- a) Es sollen nur standardisierte Prozeßvariable erlaubt sein, d.h. die Definition von Prozeßvariablen durch den Benutzer ist ausgeschlossen.
- b) Der Vorrat der standardisierten Prozeßvariablen ist um die Ein- und Ausgabemöglichkeit eines Bitmusters von 16 Bit (2 Byte) Breite zu erweitern. Der Datentyp einer derartigen Variablen sei <string>.

Für den Fall a) verlangt BASEX die Programmierung des erforderlichen Unterprogramms im Rahmen der Equipment-Anweisung direkt in Maschinencode. Durch die Möglichkeit, in BASEM bei Verwendung der CALL-Anweisung entsprechende Unterprogramme aufzurufen, erscheint die obige Einschränkung vertretbar.

Der Vorschlag b), Prozeßvariable des oben genannten Typs - nachfolgend durch den Namen IN\$ bzw. OUT\$ gekennzeichnet - einzuführen, erlaubt die direkte Bearbeitung von Meßwerten und Prozeßzuständen ohne Benutzung zusätzlicher Konversionsroutinen im Rahmen der String-Ausdrücke. Folgende Statements sind dann z.B. möglich:

```
10 LET  A$  =  IN$(K)
20 IF    B$  =  IN$(L) THEN  NN
30 LET  OUT$(K)=  IN$(L) XOR  IN$(M).
```

Gleichzeitige Bearbeitung verschiedener Programmtücke

Mikrorechner besitzen ein breites Anwendungsspektrum und sind zudem relativ preisgünstig [4]. Der Fall, daß ein einziger Mikrorechner simultan mehrere Benutzerprogramme mit gegenseitiger Beauftragung und Verwaltung der Aufträge (Multiprogramming) abarbeiten würde, wäre keinesfalls die Regel, sondern nur zur Lösung einer speziellen Aufgabe denkbar. Deshalb erschien es nicht erforderlich, daß in BASEX zum Start von Programmteilen verschiedener Priorität vorgeschlagene START-Statement in den erforderlichen Sprachumfang zur Programmierung von Mikrorechnern zu übernehmen.

4.3.2 Implementierter Sprachumfang

Dieser Abschnitt beschreibt die vorgeschlagene Semantik und Teile der Syntax des Sprachumfanges von "BASEX FOR MICROPROCESSORS"¹, abgekürzt "BASEM". Eine genaue Syntaxdefinition ist aus dem Abschnitt 4.3.3 ersichtlich. Zur Kennzeichnung der Sprachelemente wurden weitgehend die aus der Beschreibung der Programmiersprache BASEX [47] bekannten Bezeichnungen übernommen. Die Implementierung der Sprachelemente ist weitgehend abgeschlossen.

4.3.2.1 *Vorbemerkungen zur Struktur eines Programms*

BASIC ist eine zeilenorientierte Sprache. Diese Eigenschaft gilt auch für die entsprechende Real-Time Erweiterung BASEX oder die BASEX-Modifikation BASEM.

Ein BASEM Programm besteht aus einer Folge von Anweisungen und wird durch die END-Anweisung abgeschlossen. Am Anfang jeder Programmzeile steht eine unterschiedliche Zeilennummer, gefolgt von der eigentlichen Anweisung (<stat.-body>), welche mit einem "Schlüsselwort" (Anweisungsnamen) beginnt. Eine Zeile des Programms darf bei der vorliegenden Implementierung maximal 69 Zeichen beinhalten und wird mit der Eingabe eines Zeilenendekennzeichens in der Form des Steuerzeichens "Wagenrücklauf" (Carriage Return, abgekürzt CR) abgeschlossen.

4.3.2.2 *Zeichenvorrat der Sprache*

Der Zeichenvorrat umfaßt alle Buchstaben des Alphabets, die zehn Ziffern und folgende Sonderzeichen:

>	größer-als-Zeichen	+	plus-Zeichen
<	kleiner-als Zeichen	,	Komma
*	Stern	(	linke Klammer
/	Schrägstrich	)	rechte Klammer

¹ Aus Standardisierungsgründen wurde auf die Einführung einer vollständig neuen Sprachkonstruktion verzichtet

≠	ungleich-Zeichen	.	Punkt
%	Prozent-Zeichen	"	Anführungsstriche
=	Gleichheitszeichen	\$	Dollar-Zeichen
-	minus-Zeichen	␣	Leerzeichen (Space)

Zusätzliche Steuerzeichen nach Tab. 3 werden zur Steuerung des Systems eingesetzt und gehören nicht zum Zeichenvorrat der Sprache. Ihre Verwendung in Kommentaren und Strings ist im Gegensatz zu den Sonderzeichen nicht erlaubt.

4.3.2.3 Konstanten

Es sind die Datentypen <real> und <string> zugelassen, so daß Konstanten beiderlei Typs als Sprachelemente vorkommen.

`<const.> :: = <real-const.> | <string-const.>`

Real-Konstanten:

Real-Konstanten repräsentieren positive und negative Zahlen im Bereich

$$- 1.70136 \cdot 10^{38} \leq x \leq + 1.70136 \cdot 10^{38}$$

und beanspruchen systemintern einen Speicherplatz von drei Byte, wobei zwei Byte für die Mantisse und ein Byte für den Exponenten reserviert sind [26].

String-Konstanten

Es wird zwischen zwei Arten von String-Konstanten unterschieden, den ASCII-String-Konstanten und den Hexa-String-Konstanten.

`<string-const.> :: = <ascii-string-const.> |
 <hexa-string-const.>`

```
<ascii-string-const.> ::= " <ascii-character>"
<hexa-string-const.>  ::= % <hexa-character> %
```

Die Speicherung der ASCII-String-Konstanten erfolgt intern mit einem Zeichen pro Byte im ASCII-Code. Ihre Länge darf 255 Zeichen nicht übersteigen.

Bei Hexa-String-Konstanten werden intern zwei Hexa-Ziffern in einem Byte abgelegt, wobei maximal acht Hexa-Ziffern zulässig sind.

Beispiele für Konstanten

```
1    500    -21.    -.009    2.1E 5    0.99E-12
"XYZ"    "X - 3B2"    "FEHLER 5!"
%2F%    %02F4AA9E%    %E%
```

4.3.2.4 Variable

Es ist wiederum zwischen Variablen vom Typ <real> und Variablen vom Typ <string> zu unterscheiden.

```
<variable>          ::= <real-var.> | <string-var.>

<real-var.>          ::= <real-var.-name> {( <expr.> {, <expr.> }06 ) }01
<string-var.>        ::= <string-var.-name> {( <expr.> {, <expr.> }01 ) }01
```

Der Name einer Variablen besteht aus einer Zeichenfolge von maximal sechs Buchstaben und Ziffern, wobei das erste Zeichen ein Buchstabe sein muß. Dieser Variablenname darf nicht mit Funktionsnamen oder reservierten Wörtern identisch sein.

```
<real-var.-name>    ::= <letter> { <letter> | <digit> }05
<string-var.-name> ::= <real-var.-name> $
```

Real-Variable

Real-Variable kennzeichnen Speicherplätze für Gleitkommazahlen. Die Indextliste der indizierten Real-Variablen (Felder) besteht aus einen bis sieben Indizes, die durch Kommata zu trennen sind. Einfache und indizierte Real-Variable gleichen Namen sind erlaubt. Zwischen derartigen Variablen besteht keine Beziehung. Es ist jedoch nicht zulässig, gleiche Namen für indizierte Variable unterschiedlicher Dimension zu verwenden. Eine explizite Deklaration der Variablen unter Verwendung der DIM-Anweisung ist nicht erforderlich, sofern es sich um einfache Variable oder um eindimensionale Felder mit maximal zehn Feldelementen handelt.

String-Variable

Stringvariable kennzeichnen Speicherplätze, die Zeichenketten enthalten. Der Name einer String-Variablen wird am Ende durch ein zusätzliches Dollar-Zeichen \$ abgeschlossen.

Einer String-Variablen, die in keiner CHAR-Anweisung vorkommt, wird programmintern gemäß Definition 4.2 ein Speicherplatz von zwei Zeichen reserviert. Falls String-Variable aus einer größeren Anzahl von Zeichen bestehen, sind sie mit Hilfe der CHAR-Anweisung zu deklarieren, wobei die maximale Länge einer String-Variablen 255 Zeichen beträgt.

Teilstrings [47] blenden einen (zusammenhängenden) Teil aus einem String aus. Dabei bedeutet TEST\$(I,J), ein aus J Zeichen bestehender und dem I-ten Zeichen beginnender Teilstring der String-Variablen TEST\$, der vollständig in dem String enthalten sein muß.

Beispiele für Variable

X	MOTOR1	FELD(4)	FELD(A*B(Y-Z),X+Y/2)
A\$	FALL1\$		

4.3.2.5 Ausdrücke

Ausdrücke werden durch die Verknüpfung von Operanden eines Datentyps mit Operatoren gebildet. Je nach Art des Operandentyps unterscheidet man zwischen *Real-Expressions* und *String-Expressions*.

$\langle \text{expr.} \rangle :: = \langle \text{real-expr.} \rangle \mid \langle \text{string-expr.} \rangle$

Die Auswertung der Ausdrücke erfolgt von links nach rechts unter Berücksichtigung der Priorität der Operatoren, wie sie durch die Syntaxdefinition festgelegt wurde. Die Schreibweise der Ausdrücke folgt dabei den allgemeinen arithmetischen Regeln. Außerdem bindet der Test auf "gleich" oder "ungleich" schwächer, als die übrigen Vergleichsoperatoren, damit Ausdrücke der Form $A < B = C \geq D$ ohne zusätzliche Klammern geschrieben werden können. Gemäß der üblichen Vereinbarung besitzt der Operator AND eine höhere Priorität, als die Operatoren OR und XOR. Unäre Operatoren zeichnen sich durch die höchste Priorität aus.

Für logische Größen gilt ähnlich wie bei BASEX die Festlegung:

Wert = 0 $\triangleq$ "false"

Wert $\neq$ 0 $\triangleq$ "true"

String-Expressions

Entsprechend der Definition 4.3, ist in diesen Fällen die Wirkungsweise von Operatoren ausschließlich auf die ersten zwei Zeichen der Zeichenkette beschränkt. Strings der Länge ein Byte werden dann intern auf zwei Byte ergänzt (MSB = 0H).

Das erzielte Ergebnis der Verknüpfung besitzt jeweils die Länge zwei Byte.

Ausgenommen von dieser Verfahrensweise sind die logischen Operatoren. In diesen Fällen wird jeweils die gesamte Zeichenkette bitweise bearbeitet. Ist bei den logischen Operationen die Länge der zu verknüpfenden Strings nicht identisch, so bezieht sich die Verknüpfung nur auf die Länge der kürzeren Zeichenkette. Der resultierende String enthält dann außer dem Verknüpfungsergebnis der Operation unverändert die restlichen Zeichen der längeren Zeichenkette.

Für den Fall der *String-Zuweisung*

LET <string-var.> :: = <string-expr.>

gelten entsprechende Vereinbarungen wie in [48].

Arithmetische Standardfunktionen

BASIC gestattet die Verwendung sogenannter Standardfunktionen. Unter diesem Begriff versteht man mathematische Funktionen, z.B. SIN(X), LOG(X), SQR usw., die bereits im System implementiert sind, durch einen "reservierten" Namen aufgerufen und unter diesem Namen in den verschiedenen BASIC-Standards [72, 87] aufgeführt werden. Zusätzlich zu diesen Standardfunktionen gestattet BASEM den Aufruf weiterer mathematischer Funktionen, nämlich

IFX(X)	Ganzzahlteil	gerundet
und LGA(X)	dekadischer	Logarithmus $\log_{10} x$.

Als Argument sind Konstanten, Variable und Ausdrücke des Typs <real> erlaubt. Weitere Angaben zu den implementierten Standardfunktionen findet man bei [26].

Beispiele für Ausdrücke

3*X - Y**Z	SIN(X*Y/(A(I)-Z))
- X/Y	B\$*C\$-D\$

4.3.2.6 Anweisungsnummer

Anweisungsnummer (Zeilennummern, Labels) sind ganze, positive Zahlen zwischen 1 und 65 535. Die Angabe einer identischen Zeilennummer bei der Ein-

gabe einer Zeile bewirkt die Löschung des zuvor unter der äquivalenten Zeilennummer gespeicherten Quelleninformation (Zwischencode).

4.3.2.7 Nichtausführbare Anweisungen

Hierunter versteht man alle diejenigen Sprachmittel (Anweisungen), die zur Übersetzung eines vorgegebenen Quellenprogramms unbedingt erforderlich sind, bzw. zur "Lesbarkeit" des Quellenprogrammes beitragen, aber selbst nicht in Maschinenbefehle umgesetzt werden. In BASEX stehen für derartige Zwecke die Anweisungen CHAR, DATA, DEF, DIM und REM zur Verfügung. BASEM weist folgende semantische bzw. syntaktische Modifikationen auf:

- Die in einer CHAR-Anweisung definierbare maximale Stringlänge beträgt 255 Zeichen. String-Variable, die in keiner CHAR-Anweisung auftreten, werden implizit mit dem Index 2 deklariert.
- Die DEF-Anweisung erlaubt zusätzlich zur Definition von Funktion des Datentyps <real>, die Definition von String-Funktionen gemäß der angegebenen Syntax.
- In einer DIM-Anweisung können Felder mit maximal sieben Indizes deklariert werden. Der jeweilige Indexbereich beginnt bei Null und besteht somit z.B. bei einem eindimensionalen Feld mit maximalem Index N aus N + 1 Feldelementen. Explizite Deklarationen von eindimensionalen Feldern mit maximal zehn Feldelementen in einer DIM-Anweisung sind nicht erforderlich.

Ein Sprung auf eine nichtausführbare Anweisung bewirkt die Fortsetzung des Programmes bei der nächsten folgenden ausführbaren Anweisung.

4.3.2.8 Ausführbare Anweisungen

Ausführbare Anweisungen werden von dem Übersetzer in Maschinencodeform umgesetzt. Abgesehen von den erforderlichen Modifikationen hinsichtlich der String-Expressions, sind gegenüber BASEX folgende Anweisungen bzw. Anweisungselemente geringfügig modifiziert:

FOR-Anweisung

Entsprechend den BASIC-Standards, erlaubt BASEM als Laufvariable nur einfache Variable.

```
<for-st.>          ::= FOR <simple-ass.-expr.> TO <expr.>
                      {STEP <expr.>}01
<simple-ass.-expr.> ::= <simple-real-var.>  = <real-expr.>|
                      <simple-string-var.> = <string-expr.>
```

Sprünge in eine FOR-Schleife oder aus dieser heraus sind zulässig, sollten jedoch aus Gründen der übersichtlichen Programmierung vermieden werden. Es sei an dieser Stelle vermerkt, daß besonders bei der Programmierung der FOR-Schleifen durch Verwendung der String-Expressions ein erheblicher Rechenzeitgewinn zu erzielen ist.

INPUT-Anweisung

Die INPUT-Anweisung gestattet dem Benutzer die interaktive Zuweisung von Daten an Variable während des Programmlaufes. Die Möglichkeit der Ausgabe einer ASCII-String-Konstanten auf dem Konsolgerät wurde um die Variante der Ausgabe von Hexa-String-Konstanten erweitert. So lassen sich nun u.a. auch Steuerzeichen ausgeben, z.B. BELL um den Benutzer auf eine folgende Eingabe aufmerksam zu machen.

```
<input-st.> ::= INPUT {<dev.-option>|<string-const>|
                      <variable>} {,<dev.-option>|,<string-const>|
                      ,<variable>}0*
```

Auf Grund dieser festgelegten Syntax des INPUT-Statements, besteht keine weitergehende Beschränkung hinsichtlich der Anordnung einzelner Satz-elemente.

Die Eingabe erfolgt im Normalfall über das Konsolgerät. Wird ein anderes Eingabegerät gewünscht, so ist dies durch die Angabe einer systemspezifischen Gerätenummer n bei Verwendung des "Steuerwortes" DEV(n) möglich (Ausnutzung der Alternative <dev.-option>). Eine derartige Zuordnung besteht dann bis zur erneuten Spezifizierung für alle folgenden Eingabevorgänge [z.B. 45, 48].

PRINT-Anweisung

Die PRINT-Anweisung ermöglicht die Ausgabe von Zahlenwerten und Texten auf einem Ausgabegerät. Dieses Ausgabegerät kann ähnlich wie bei der INPUT-Anweisung anhand des "Steuerwortes" DEV(n) zusätzlich spezifiziert werden.

Sowohl in BASIC [72, 87] als auch in der RT-Erweiterung BASEX [47], sind als Separatoren zwischen den PRINT-Elementen das Komma und das Semikolon zugelassen. Dabei wurde festgelegt, daß nach einem Komma die Ausgabe der folgenden Information erst in der nächsten "Tabulatorspalte" beginnt, nach einem Semikolon jedoch unmittelbar folgend auf die vorhergehende Ausgabeinformation.

Das Semikolon würde nur im Rahmen der PRINT-Anweisung benötigt, so daß bei BASEM im Hinblick auf Kleinsysteme (Einsparung von Operatoren), nur ausschließlich das Komma als Separator dient.

$$\begin{aligned} \text{<print-st.>} &:: = \text{PRINT } \text{<print-element>} \{, \text{<print-element>}\}^* \{, \}^1_0 \\ \text{<print-element>} &:: = \text{<device-option>} | \text{<tab.-option>} | \text{<expr.>} \end{aligned}$$

Jede Ausgabe schließt sich nun - bis auf ein Leerzeichen - direkt an die vorhergehende an, sofern von der Möglichkeit der Positionierung einer Ausgabe durch Verwendung des "Steuerwortes" TAB(x) kein Gebrauch gemacht wurde.

In BASEM erfolgt die Ausgabe von Zahlen des Datentyps <real> in einem "automatischen" Format [37], so daß die Spezifikation <format-option> in der Syntax der PRINT-Anweisung nicht erscheint.

4.3.2.9 Sprachelemente für Anforderungen der Prozeßdatenverarbeitung

Nach Berücksichtigung der in Abschnitt 4.3.1 vorgeschlagenen Modifikationen, konnten für diesen Aufgabenbereich im wesentlichen die in [48] vorgestellten Erweiterungen für die Implementierung bei Mikrorechnern herangezogen werden. In Abschnitt 4.3.1 erfolgte eine ausführliche Besprechung der gewünschten Modifikationen, so daß an dieser Stelle eine kurze, zusammenfassende Darstellung der Semantik und Syntax dieser Elemente möglich ist.

"Standardisierte" Prozeßvariable

Zum Datenaustausch mit Prozessen sind in BASEM die nachfolgend angeführten Prozeßvariablen als intern "fest" vereinbart. Nähere Angaben hierzu findet man in Kapitel 5.2 oder bei [42]. Als Parameter x dieser Prozeßvariablen ist ein Ausdruck erlaubt.

- | | | | |
|----|----------|----------|---|
| a) | INB(x), | OUTB(x) | Ein-/Ausgabe eines Bits |
| b) | INW(x), | OUTW(x) | Ein-/Ausgabe eines 16-Bit Wortes, |
| c) | IND(x), | OUTD(x) | Ein-/Ausgabe einer vierstelligen BCD-Zahl |
| d) | IN\$(x), | OUT\$(x) | Ein-/Ausgabe eines Bitmusters von 16 Bit Breite |

In den Fällen a) bis c) wird im Bearbeitungsfall gleichzeitig ein Konversion in das interne Gleitkommaformat bzw. von der Gleitkommadarstellung in das gewünschte Ausgabeformat durchgeführt. Im Fall d) dagegen erfolgt eine Übernahme und Speicherung des entsprechenden Bitmusters als String der Länge zwei Byte. Die Prozeßvariable OUT\$ gestattet die Ausgabe der ersten zwei Byte eines Strings.

Call-Anweisung

Die Call-Anweisung ermöglicht den Anschluß eines bereits in Maschinen-code vorliegenden Unterprogrammes, welches eventuell für spezielle Aufgaben von dem Anwender in Assembler geschrieben und bereits assembliert wurde.

$$\langle \text{call-st.} \rangle ::= \text{CALL } \langle \text{subr.-name} \rangle \{ \{ \langle \text{expr.} \rangle \{ \langle \text{expr.} \rangle \}^2 \}_0 \}$$

Es ist die Angabe von maximal drei Parametern zulässig. Diese Zahl wurde durch die Systemstruktur des betrachteten Mikroprozessors *INTEL 8080* bestimmt. Da dieser Prozessor drei Registerpaare für derartige Aufgaben zur Verfügung stellt, kann die folgende Übergabeart festgelegt werden:

Der Übersetzer übergibt bzw. erwartet die Adressen der jeweiligen Parameter (siehe auch Abschnitt 5.1.3) in den vorhandenen Registerpaaren. Dabei gilt die Festlegung:

Adresse des ersten Parameters: BC-Registerpaar

Adresse des zweiten Parameters: DE-Registerpaar

Adresse des dritten Parameters: HL-Registerpaar

Jedes von dem Anwender geschriebene Unterprogramm ist wie üblich mit RETURN abzuschließen.

After-Anweisung

Die AFTER-Anweisung bewirkt einen Programmstopp von definierter zeitlicher Länge. Die Zeit in Millisekunden, nach deren Ablauf die folgende Anweisung abgearbeitet wird, bestimmt der Ausdruck hinter dem Schlüsselwort AFTER.

$$\langle \text{after-st.} \rangle ::= \text{AFTER } \langle \text{expr.} \rangle, \langle \text{op.-1-st.} \rangle$$

Der in BASEX [47,48] vorgesehene Operator ":" wurde ebenso wie in der ONINT-Anweisung durch "," ersetzt, um im Hinblick auf Kleinstsysteme wiederum Operatoren einsparen zu können. Der maximale Wert des Ausdrucks ist vorläufig auf 65535 Millisekunden begrenzt.

Anweisungen zur Sperrung und Freigabe des Interruptsystems

Es ist eine allgemeine Sperrung oder Freigabe von Unterbrechungen mit Hilfe der Anweisungen DISAB oder ENAB möglich.

`<disable-st.> :: = DISAB, <enable-st.> :: = ENAB`

Die Möglichkeit, Unterbrechungen abhängig von der zugeordneten Interrupt-Nummer freizugeben oder zu verhindern, würde bei dem betrachteten Mikroprozessor zusätzliche Hardware erfordern

ONINT-Anweisung

Die ONINT-Anweisung gestattet die asynchrone Bearbeitung von Anweisungen bzw. Interruptprogrammen.

`<onint-st.> :: = ONINT <interrupt-no.>, <op.-1-st.>`

Ein mit der Anweisung GOSUB erreichbares Interruptprogramm ist durch die Anweisung RETURN abzuschließen.

Die Art der Interruptbehandlung ist bei Mikroprozessoren sehr unterschiedlich. So bearbeitet z.B. das betrachtete System INTEL 8080 nach Erkennung eines Interruptes, einen asynchron mit dem "Interrupt-Acknowledge-Signal" auf den Datenbus geschalteten Befehlscode [12]. Dies kann jeder beliebige Befehl sein. Um jedoch in den normalen Programmablauf nach Abarbeitung des gewünschten Interruptprogrammes wieder zurückzufinden, wird der Anwender für diesen Zweck vermutlich jenen Typ von Befehlen hardwaremäßig "einschleusen", welcher vor Beginn der Abarbeitung die Speicherung der Unterbrechungsadresse gewährleistet, also einen Call- oder den spezielleren RESTART-Befehl.

Diese Befehle beinhalten gewünschte Unterprogrammadressen in direkter bzw. verschlüsselter Form. Dem Anwender müssen somit zur Vorbereitung der Hardware die endgültigen Adressen eventueller Interruptprogramme mitgeteilt werden. Nach der Durchführung des Übersetzungsvorganges erfolgt deshalb der Ausdruck aller im Quellenprogramm vorhandener Interruptnummern mit den zu-

geordneten Objektcodeadressen der jeweiligen Interruptprogramme. Durch dieses Konzept ist eine weitgehende Systemunabhängigkeit möglich.

Stop-Anweisung

Die Ausführung der STOP-Anweisung veranlaßt die Überführung des Systems in den HALT-Zustand. Zuvor erfolgt die Speicherung aller Registerinhalte und Statusinformationen. Der HALT-Zustand kann nur nach einem Interrupt verlassen werden.

`<stop-st.> :: = STOP`

WAIT-Anweisung

Nach Erreichen dieser Anweisung läuft das Programm in einer internen Programmschleife, die fortwährend den nach dem "Schlüsselwort" WAIT stehenden Ausdruck ermittelt. Sobald der Wert dieses Ausdruckes ungleich Null ist (z.B. durch Änderung einer Prozeßvariablen), wird das Programm mit der folgenden Anweisung fortgeführt.

`<wait-st.> :: = WAIT <expr.>`

4.3.3 Syntax der BASEX-Modifikation BASEM

In diesem Abschnitt wird die Syntax der Programmiersprache BASEM in EBNF vorgestellt. Die verwandten Methasymbole entsprechen den aus der Literatur [z.B. 88, 93] bekannten Formen. Dabei steht für:

`:: =` Definitionszeichen einer Produktionsregel
`< >` Begrenzung einer Meta-Variablen
`|` Trennzeichen für Alternativen einer Produktion
`{ }ij` Mögliche Wiederholungen
 `{ }ab` $0 \leq a \leq b$, a,b sind natürliche Zahlen
`{ }` Abkürzung für `{ }11`
`{ }a*` "*" ist Abkürzung für: "beliebig oft"

Syntax der BASEX-Modifikation BASEM in EBNF

1. S :: = <statement> CR | <command> CR
2. <command> :: = @{<comm.-1>|<dump.-comm.>|<set-comm.>|<time-comm.>}
- 2.1 <comm.-1> :: = <comm.-1-name>{<comm.-spec.-1>}₀¹
- 2.2 <comm.-1-name> :: = ASSIGN|CALCULATE|COMPILE|DELETE|DISPLAY|END|HEXPUNCH|INFORM|IMMEDIATE|LIST|NEW|OLD|PUNCH|READ|RENUM|RUN|START|TRANSFER
- 2.3 <comm.-spec.-1> :: = {<comm.-spec.-1-element>}₀¹
{,<comm.-spec.-1-element>}₀²
- 2.4 <comm.-spec.-1-element> :: = {<digit>}₀⁵|<digit>{<hexa>}₀³H|0{<hexa>}₀⁴H
- 2.5 <dump.-comm.> :: = DUMP <comm.-var.>{,<comm.-var.>}₀^{*}
- 2.5.1 <comm.-var.> :: = <num.-comm.-var.>|<string-comm.-var.>
- 2.5.2 <num.-comm.-var.> :: = <num.-var.-name>{(<num.-const.>{,<num.-const.>}₀⁶)}₀¹
- 2.5.3 <string-comm.-var.> :: = <string-var.-name>{(<num.-const.>{,<num.-const.>}₀¹)}₀¹
- 2.6 <set-comm.> :: = SET {<num.-comm.-var.> = <num.-const.>|<string-comm.-var.> = <string-const.>|{,<num.-comm.-var.> = <num.-const.>|<string-comm.-var.> = <string-const.>}₀^{*}}

- 2.7 <time-comm.> :: = TIME {<hour-spec.>{:<min.-spec.>{:<sec.-spec.>}₀}₀}₀^{1 1 1}
3. <statement> :: = <stat.-no> <stat.-body>
- 3.1 <stat.-body> :: = <decl.-1-st.>|<decl.-2-st.>|<op.-1-st.>|<op.-2-st.>
- 3.2 <decl.-1-st.> :: = <char.-st.>|<data-st.>|<def.-st.>|<dim.-st.>|<rem-st.>
- 3.3 <decl.-2-st.> :: = <after-st.>|<onint-st.>
- 3.4 <op.-1-st.> :: = <call-st.>|<end-st.>|<gosub-st.>|<goto-st.>|<let-st.>|<stop-st.>
- 3.5 <op.-2-st.> :: = <comp.-gosub-st.>|<comp.-goto-st.>|<disable-st.>|<enable-st.>|<for-st.>|<if-st.>|<input-st.>|<next-st.>|<print-st.>|<read-st.>|<restore-st.>|<return-st.>|<wait-st.>
- 3.6 <char.-st.> :: = CHAR <char.-element>{,<char.-element>}₀^{*}
- 3.6.1 <char.-element> :: = <string-var.-name>(<real-const.>)
- 3.7 <data-st.> :: = DATA <const.>{,<const.>}₀^{*}
- 3.8 <def.-st.> :: = <real-def.-st.>|<string-def.-st.>
- 3.8.1 <real-def.-st.> :: = DEF <users-real-fct.-name>(<real-var.-name>) = <real-expr.>
- 3.8.2 <string-def.-st.> :: = DEF <users-string-fct.-name>(<string-var.-name>) = <string-expr.>
- 3.9 <dim.-st.> :: = DIM <dim.-element>{,<dim.-element>}₀^{*}
- 3.9.1 <dim.-element> :: = <real-var.-name>(<real-const.>{,<real-const.>}₀⁶)
- 3.10 <rem-st.> :: = REM {<alphanumeric>}₀^{*}

3.11	<after-st.>	:: = AFTER <expr.>,<op.-1-st.>
3.12	<onint-st.>	:: = ONINT <interrupt-no.>,<op.-1-st.>
3.12.1	<interrupt-no.>	:: = <real-const.>
3.13	<call-st.>	:: = CALL <subr.-name>{(<expr.>{,<expr.> ₀ ² }} ₀ ¹
3.14	<end-st.>	:: = END
3.15	<goto-st.>	:: = GOTO <stat.-no.>
3.16	<let-st.>	:: = LET <ass.-expr.>{,<ass.-expr.>} ₀ [*]
3.17	<stop-st.>	:: = STOP
3.18	<comp.-gosub-st.>	:: = GOSUB <expr.> OF <stat.-no.>{,<stat.-no.>} ₀ [*]
3.19	<comp.-goto-st.>	:: = GOTO <expr.> OF <stat.-no.>{,<stat.-no.>} ₀ [*]
3.20	<disable-st.>	:: = DISAB
3.21	<enable-st.>	:: = ENAB
3.22	<for-st.>	:: = FOR <simple-ass.-expr.> TO <expr.>{STEP<expr.>} ₀ ¹
3.23	<gosub-st.>	:: = GOSUB <stat.-no.>
3.24	<if-st.>	:: = IF <expr.> THEN <stat.-no.>
3.25	<input-st.>	:: = INPUT {<dev.-option> <string-const.> <variable>} {,<dev.-option> ,<string-const.> ,<variable>} ₀ [*]
3.26	<next-st.>	:: = NEXT <simple-var.>
3.27	<print-st.>	:: = PRINT <print-element>{,<print-element>} ₀ [*] {,} ₀ ¹

3.27.1	<print-element>	:: = <expr.> <tab.-option> <device-option>
3.28	<read-st.>	:: = READ <variable>{,<variable>} ₀ *
3.29	<restore-st.>	:: = RESTORE
3.30	<return-st.>	:: = RETURN
3.31	<wait-st.>	:: = WAIT <expr.>
4.	<ass.-expr.>	:: = <real-ass.-expr.> <string-ass.-expr.>
4.1	<real-ass.-expr.>	:: = <real-var.> = <real-expr.> <real-process-out.-var.> = <real-expr.>
4.2	<string-ass.-expr.>	:: = <string-var.> = <string-expr.> <string-process-out.-var.> = <string-expr.>
4.3	<simple-ass.-expr.>	:: = <simple-real-var.> = <real-expr.> <simple-string-var.> = <string-expr.>
4.4	<expr.>	:: = <real-expr.> <string-expr.>
4.5	<variable>	:: = <real-var.> <string-var.>
4.6	<const.>	:: = <real-const.> <string-const.>
4.7	<real-expr.>	:: = <real-term-1> <real-expr.><log-1-op.><real-term-1>
4.7.1	<real-term-1>	:: = <real-term-2> <real-term-1><log-2-op.><real-term-2>
4.7.2	<real-term-2>	:: = <real-term-3> <real-term-2><rel.-1-op.><real-term-3>
4.7.3	<real-term-3>	:: = <real-term-4> <real-term-3><rel.-2-op.><real-term-4>
4.7.4	<real-term-4>	:: = <real-term-5> <real-term-4><sond.-op.><real-term-5>

4.7.5	<real-term-5>	:: = <real-term-6> <real-term-5><add.-op.><real-term-6>
4.7.6	<real-term-6>	:: = <real-term-7> <real-term-6><mul.-op.><real-term-7>
4.7.7	<real-term-7>	:: = <real-oprd.> <real-term-7><exp.-op.><real-oprd.>
4.7.8	<real-oprd.>	:: = <unary-op.><real-oprd.> <real-var.> <real-const.> (<real-expr.>) <std.-fct.> <real-users-fct.> <real-process-input-var.> CHG (<string-expr.>)
4.8	<string-expr.>	:: = <string-term-1> <string-expr.><log.-1-op.> <string-term-1>
4.8.1	<string-term-1>	:: = <string-term-2> <string-term-1><log.-2-op.> <string-term-2>
4.8.2	<string-term-2>	:: = <string-term-3> <string-term-2><rel.-1-op.> <string-term-3>
4.8.3	<string-term-3>	:: = <string-term-4> <string-term-3><rel.-2-op.> <string-term-4>
4.8.4	<string-term-4>	:: = <string-term-5> <string-term 4><sond.-op.> <string-term-5>
4.8.5	<string-term-5>	:: = <string-term-6> <string-term-5><add.-op.> <string-term-6>
4.8.6	<string-term-6>	:: = <string-term-7> <string-term-6><mul.-op.> <string-term-7>
4.8.7	<string-term-7>	:: = <string-oprd.> <string-term-7><exp.-op.><string-oprd.>
4.8.8	<string.oprd.>	:: = <unary-op.><string-oprd.> <string-variable> <string-const.> (<string-expr.>) <string-user-fct.> <string-process-input-var.> CHG (<real-expr.>)

4.9	<real-var.>	:: = <real-var.-name>{(<expr.>{, <expr.> } ₀ ⁶) } ₀ ¹
4.10	<simple-real-var.>	:: <real-var.-name>
4.11	<string-var.>	:: <string-var.-name>{(<expr.>{, <expr.> } ₀ ¹) } ₀ ¹
4.12	<simple-string-var.>	:: <string-var.-name>
4.13	<simple-var.>	:: = <real-var.-name>'<string-var.-name>
4.14	<string-const.>	:: = <ascii-string-const.> <hexa-string-const.>
4.15	<real-process-input-var.>	:: = <real-process-input-var.-name>(<expr.>)
4.16	<string-process-input-var.>	:: = <string-process-input-var.-name>(<expr.>)
4.17	<real-process-out.-var.>	:: <real-process-out.-var.-name>(<expr.>)
4.18	<string-process-out.-var.>	:: <string-process-out.-var.-name>(<expr.>)
4.19	<real-user-fct.>	:: = <real-user-fct.-name>(<real-expr.>)
4.20	<string-user-fct.>	:: <string-user-fct.-name>(<string-expr.>)
4.21	<std.-fct.>	:: <std.-fct.-name>(<real-expr.>)
4.22	<string.-var.-name>	:: <real-var.-name> \$
4.23	<subr.-name>	:: <real-var.-name>
4.24	<real-process-input-var.-name>	:: INB IND INW INWD
4.25	<string-process-input-var.-name>	:: IN\$ INW\$
4.26	<real-process-out.-var.-name>	:: = OUTB OUTD OUTW OUTWD
4.27	<string-process-out.-var.-name>	:: = OUT\$ OUTW\$

4.28	<real-user-fct.-name>	:: = FN<real-var.-name>
4.29	<string-user-fct.-name>	:: = FN<string-var.-name>
4.30	<std.-fct.-name>	:: = ABS ATN COS EXP IFX INT LGA LOG RND SGN SIN SQR TAN
4.31	<real-var.-name>	:: = <letter>{<letter> <digit> ₀ ⁵ }
4.32	<real-const.>	:: = {+ -} ₀ ¹ <dec.-fraction>{E{+ -} ₀ ¹ {<digit> ₁ ² } ₀ ¹ }
4.33	<dec.-fraction>	:: = {<digit> ₁ ⁵ } {<digit>}.{<digit>}(max. 5 Ziffern)
4.34	<ascii-string-const.>	:: = "{<alphanumeric> ₁ *"}"
4.35	<hexa-string-const.>	:: = %<hexa> ₁ ⁸ %
4.36	<log.-1-op.>	:: = XOR OR
4.37	<log.-2-op.>	:: = AND
4.38	<rel.-1-op.>	:: = ≠ =
4.39	<rel.-2-op.>	:: = < > <=> >=
4.40	<sond.-op.>	:: = MAX MIN
4.41	<add.-op.>	:: = + -
4.42	<mul.-op.>	:: = *//
4.43	<exp.-op.>	:: = **
4.44	<unary-op.>	:: = + - NOT
4.45	<tab.-option>	:: = TAB (<expr.>)
4.46	<device-option>	:: = DEV (<expr.>)

4.47	<stat.-no.>	:: = {<digit>} ₁ ⁵
4.48	<hour-spec.>	:: = {<digit>} ₁ ²
4.49	<min.-spec.>	:: = {<digit>} ₁ ²
4.50	<sec.-spec.>	:: = {<digit>} ₁ ²
4.51	<alphanumeric>	:: = <digit> <letter> <special>
4.52	<hexa>	:: = <digit> A B C D E F
4.53	<digit>	:: = 0 1 2 3 4 5 6 7 8 9
4.54	<letter>	:: = A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
4.55	<special>	:: = > < * / # % = - + , () ' . ! "

5. IMPLEMENTIERUNG VON STANDARD-SYSTEMPROGRAMMEN

Real-Time BASIC spricht eine Anzahl von Systemprogrammen an, so z.B. Unterprogramme für die arithmetischen Operatoren, die Standardfunktionen oder die Unterprogramme zur Abwicklung der Prozeß Ein-/Ausgaben. Zur Implementierung der erforderlichen Unterprogramme wurde ein Konzept erarbeitet und realisiert, welches aus den folgenden Abschnitten ersichtlich ist.

5.1 Arithmetische Unterprogramme

In der Prozeßrechentechnik erfolgt die Problemlösung oft unter Anwendung mathematischer Modelle. Dies macht es erforderlich, daß der Rechner die Grundrechenarten und noch einiges mehr - z.B. die Standardfunktionen - "beherrscht". Während bei größeren Rechnersystemen derartige Aufgaben meist von der vorhandenen Hardware abgewickelt werden, können Mikrorechner das geforderte Ergebnis derzeit nur durch Einsatz zusätzlich implementierter Software erreichen.

5.1.1 Umfang der zu realisierenden Unterprogrammbibliothek

Am Anfang der Überlegungen steht die Frage, welche Unterprogramme im einzelnen zu erstellen sind, um einen möglichst großen Teil der auftretenden arithmetischen Aufgaben auf direktem Wege zu bewältigen. An erster Stelle sind hier die spezifischen arithmetischen BASIC-Operatoren und Standardfunktionen [72] zu nennen, die durch eine Anzahl von Hilfsprogrammen zur internen Organisation ergänzt werden müssen. Betrachtet man z.B. eine einfache Gleitkommaaddition, so ist im einzelnen u.a. durchzuführen:

- Angleichung der kleineren Mantisse an die größere,
- Festkommaaddition der Mantissen,
- Runden der Mantisse,
- Prüfen des Exponenten auf Überlauf und
- Wegspeichern des Resultates

Schließlich sind noch Softwaremodule zur Ein- und Ausgabe von Daten erforderlich. Tab. 1 gibt einen Überblick über die wichtigsten zu erstellenden Unterprogramme.

Library 1		Library 2	
Library 1 besteht aus		Library 2 besteht aus	
Unterprogrammen für die Festkommaarithmetik (16 Bit)		Unterprogrammen für mathematische Funktionen (Gleitkomma 24 Bit)	
Unterprogrammen für die Gleitkommaarithmetik (24 Bit)			
Unterprogrammen für die Ein-/Ausgabe			
Name	Funktion	Name	Funktion
IADD	Festkommaaddition	EXP	Exponentialfunktion e^x
ISUB	Festkommasubtraktion	LOG	natürlicher Logarithmus $\ln(x)$
IMUL	Festkommultiplikation (32 Bit Produkt)	LGA	dekadischer Logarithmus $\log_{10}(x)$
IMUL2	Festkommultiplikation (16 Bit Produkt)	SQR	Quadratwurzel $\sqrt{x}$
IDIV	Festkommdivision (32 Bit Dividend)	COS	Cosinus $\cos(x)$
IDIV2	Festkommdivision (16 Bit Dividend)	SIN	Sinus $\sin(x)$
IABS	Festkommabetrag	TAN	Tangens $\tan(x)$
INEG	Festkommanegieren	ATN	Arcus tangens $\arctan(x)$
GADD	Gleitkommaaddition	PWR	Potenzen x^y
GSUB	Gleitkommasubtraktion		
GMUL	Gleitkommultiplikation		
GDIV	Gleitkommdivision		
ABS	Gleitkommabetrag		
NEG	Gleitkommanegieren		
INT	Ganzzahlteil		
IFX	Ganzzahlteil gerundet		
SGN	Signum		
NORMA	Gleitkomma normalisieren		
STORE	Gleitkomma transportieren		
IN	Gleitkommazahl einlesen		
PRINT	Gleitkommazahl ausdrucken		
PTEXT	Text ausdrucken		

Tab.1: Verzeichnis zu erstellender Unterprogramme

Bei der Lösung eines Problems ist im allgemeinen nur ein Teil der vorgegebenen Unterprogramme beteiligt, so daß eine gewisse Unabhängigkeit der einzelnen Teile der Bibliothek gewährleistet sein muß. Dies bedingt einen modularen Aufbau der Bibliothek, wie er in Abb. 7 angedeutet wird. Somit

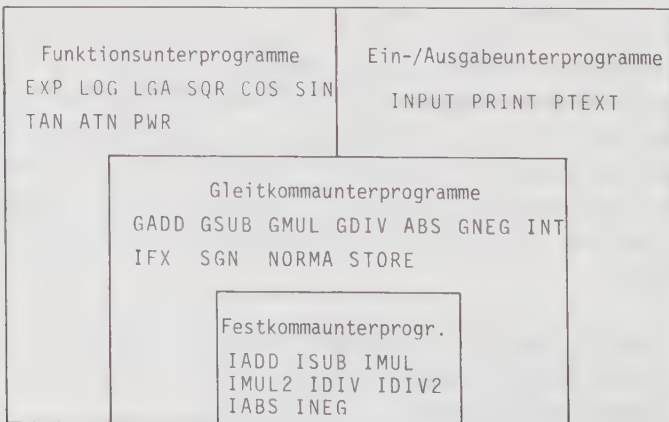


Abb.7: Modulaufbau der zu erstellenden Bibliothek

ist eine möglichst große Unabhängigkeit der einzelnen Module gewährleistet. Anhand dieser Struktur sind auch evtl. Änderungen leichter durchzuführen.

5.1.2 Auswahl des Zahlenformates

Die verschiedenen Mikrorechner besitzen jeweils spezifische Systemeigenschaften, die die Vorgehensweise der Softwareerstellung beeinflussen. Wie bereits unter Kapitel 2.1 erwähnt, ist das betrachtete "Quasi-Norm" System INTEL 8080 bytestrukturiert.

Bei der Auswahl des Zahlenformates muß berücksichtigt werden, daß sowohl Festkomma- als auch Gleitkommaoperationen durch Unterprogramme zu realisieren sind. Da verschiedene Gleitkommaprogramme von der Festkommaarithmetik Gebrauch machen, wird durch das Format der Festkommazahlen die Länge der Mantisse von Gleitkommazahlen festgelegt. Wegen der 2er-Komplement-Struktur der betrachteten CPU verwenden die Unterprogramme ebenfalls diese Form.

Die Genauigkeit der Zahlendarstellung ist bekanntlich abhängig von dem gewählten Datenformat. Auch muß beachtet werden, daß u.U. ein beträchtlicher Genauigkeitsverlust bei längeren Rechnungen eintreten kann, der seine Ursache meist in der Subtraktion zweier ungefähr gleich großer Zahlen hat. Rundungsfehler haben in der Regel einen geringeren Einfluß, da durch Anwendung einer geeigneten Rundungsvorschrift der Grenzwert des mittleren Rundungsfehlers gegen Null strebt. Einen Beweis für die Aussage findet man u.a. bei DOTZAUER [71].

Die Aufgabenstellung der Prozeßdatenverarbeitung und die vorgegebene Bytestruktur legen zur Darstellung von Festkommazahlen - wie auch in [78] angegeben - die Wahl einer Zahlenformatlänge von zwei Byte entsprechend 16 Bit nahe.

Diese Darstellung ergibt eine wesentlich höhere Genauigkeit als 10^{-6} , andererseits werden Speicher- und Zeitfaktor klein gehalten. Das gewählte Format für Festkommazahlen ist aus Abb. 8.1 ersichtlich. Der "Binärpunkt" liegt rechts von der niedrigsten Bitstelle, so daß nur ganze Zahlen im

Bereich

$$(1) \quad - 32768 \leq x \leq 32767$$

dargestellt werden.

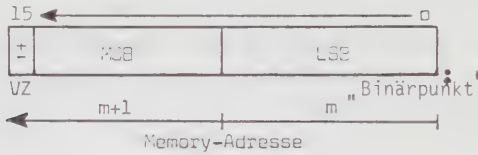


Abb. 8.1: Zahlenformat der Festkommazahlen

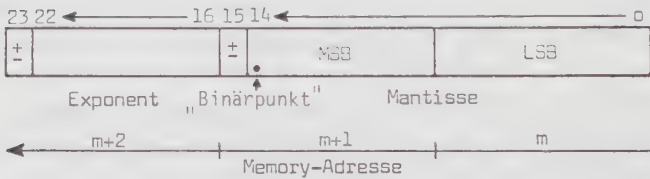


Abb. 8.2: Zahlenformat der Gleitkommazahlen

Auf Grund der obigen Überlegungen ist die Mantissenlänge bei Gleitkommazahlen auf zwei Byte festgelegt (Mantisse m). Verwendet man für den Exponenten e zusätzlich noch ein Byte [78], so sind bei Potenzen zur Basis 2 Zahlenwerte x im Bereich ("Binärpunkt" liegt zwischen Vorzeichen und hochwertiger Bitstelle)

$$(2) \quad - 1.70141 \cdot 10^{38} \leq x \leq 1.7041 \cdot 10^{38}, \quad x = m \cdot 2^e$$

darstellbar. Abb. 8.2 zeigt das gewählte Format für Gleitkommazahlen, die Darstellung erfolgt normalisiert.

Der absolute Fehler ϵ_a der Zahlendarstellung beträgt maximal - bedingt durch das Rundungsverfahren - die Hälfte des letzten Mantissenbitwertes, d.h.

$$(3) \quad \epsilon_{a_{\max}} = \frac{1}{2} \cdot 2^{-n} = 2^{-(n+1)},$$

wobei $-n$ für die Gewichtung des letzten Bits steht (siehe auch Zahlenformat nach Abb. 8). Für den relativen Fehler ergibt sich dann:

$$(4) \quad |\epsilon_{r_{\max}}| = \left| \frac{\epsilon_{a_{\max}}}{m_{\min}} \right|, \quad m_{\min} = \text{Kleinster exakter Wert der Mantisse}$$

Da die Mantisse normalisiert vorliegt, beträgt der Mantissenwert m bei dem gewählten Zahlenformat

$$0,5 \leq m < 1,0$$

so daß für den relativen maximalen Fehler der Zahlendarstellung ein Wert von

$$|\epsilon_{r_{\max}}| = \frac{\epsilon_{a_{\max}}}{m_{\min}} = \frac{2^{-(n+1)}}{1/2} = 2^{-n}$$

erreicht wird.

Für das gewählte Zahlenformat ergibt sich somit für $n = 15$

$$(5) \quad |\epsilon_{r_{\max}}| = 2^{-15} = 3,05175 \cdot 10^{-5}$$

Die Genauigkeit der Zahlendarstellung wird demnach durch den in Gl.(5) angegebenen Wert begrenzt. Da Mikrorechner auch für diejenigen Aufgabenbereiche eingesetzt werden sollen, die zuvor überwiegend von analogen Systemen bewältigt wurden, erscheint die gewählte Genauigkeit ausreichend.

Sollte jedoch für spezielle Aufgaben eine größere Genauigkeit der Zahlendarstellung erforderlich sein, so ist beispielsweise eine allgemeine Erweiterung des Mantissenformates vorzunehmen oder der Benutzer muß für

diese Spezialfälle eigene Programme anschließen. In jedem Fall wird jedoch dann eine Zunahme der Rechenzeit nicht zu vermeiden sein, sofern keine zusätzliche Hardware zur Verfügung steht, was besonders bei den derzeit noch etwas langsameren Mikroprozessoren nur bei speziellen Genauigkeitsanforderungen zu vertreten ist.

5.1.7 Struktur der Unterprogramme und Organisation des Unterprogrammaufrufes

Zur Konzeption der zu erstellenden Unterprogramme waren eine Reihe von Vorüberlegungen notwendig. Einen Einblick in die Vorgehensweise bei der Festlegung des Implementierungskonzeptes für die Unterprogrammtechnik soll am Beispiel der Parameterübergabe und des Verfahrens der Fehlerkennzeichnung gezeigt werden.

Fordert ein Unterprogramm Parameter an, so muß es nicht nur mit der Rückkehradresse, sondern auch mit Operanden selbst oder deren Adressen versorgt werden. Eine solche Parameterversorgung kann z.B. auf folgende Art erfolgen [93]:

- a) Die Information steht in einem festen Speicherbereich.
- b) Die Information steht in Registern.
- c) Ein Register enthält einen Zeiger (Adresse) auf einen Speicherbereich, in welchem die eigentliche Parameterinformation abgelegt ist.

Fall a) scheidet aus, weil er voraussetzt, daß man für alle eventuell im selben Programm verwendbaren Unterprogramme voneinander verschiedene, feste Parameter-Versorgungsplätze bereitstellt. Bei arithmetischen Operationen werden je Operator maximal zwei Operandenwerte verknüpft, d.h. neben dem Speicherplatz für die zwei zu verknüpfenden Variablen ist nur noch die Versorgung der Unterprogramme mit einer Adresse zur Abspeicherung des Ergebnisses erforderlich. Um schließlich bei der Parameterübergabe unabhängig von der Operandenlänge zu werden, erweist es sich als vorteilhaft, wenn statt der Operanden selbst, eine Übergabe der entsprechenden Adressen erfolgt.

Die gewählte Form der Parameterübergabe stellt deshalb eine Kombination von b) und c) dar, nämlich die Übergabe der Anfangsadresse des jeweiligen Parameters in festen Registern, was durch die Hardwarestruktur des betrachteten Mikroprozessors unterstützt wird. Die verschiedenen Registerpaare BC, HL und DE (Abb.9) weisen jedoch hinsichtlich ihrer Verwendungsmöglichkeit erhebliche Unterschiede auf. So muß besonders auf die Hierarchie der Registerpaare bezüglich des *register indirect addressing* geachtet werden [21].

Dieser hierarchische Aufbau ist mitbestimmend bei der Konzeption der arithmetischen Unterprogramme. Anhand von Abb.9 ist die gewählte Form der Parameterübergabe ersichtlich. Der Befehlsvorrat des betrachteten

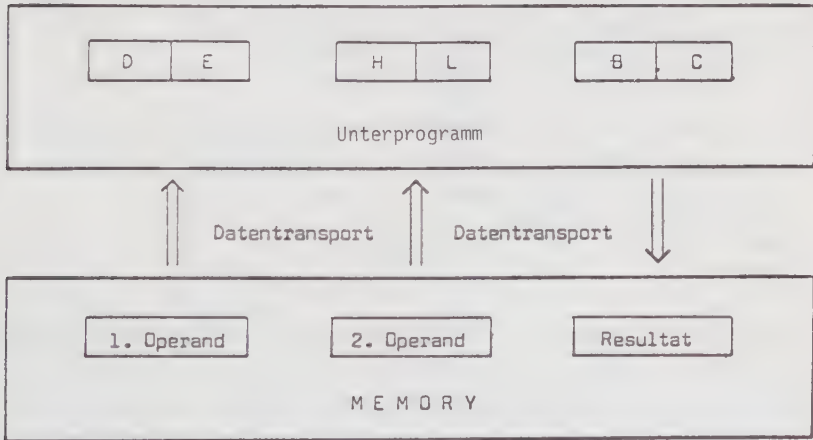


Abb.9:Prinzipielle Form der Parameterübergabe

Mikroprozessors ermöglicht dabei durch die sogenannten *load index immediate* (LXI) Befehle ein schnelles und bequemes Laden der Adressen in die Register, bzw. mit Hilfe der Befehle LDAX D bzw. MOV A,M einen leichten Zugriff zu den Operandewerten [26].

Bei der Planung eines Rechensystems ist es unabdingbar, Fehlermeldungen für die verschiedenen Fehlersituationen vorzusehen. Der damit verbundene Organisationsaufwand und die (geringfügig) höhere Rechenzeit werden von dem Benutzer gern in Kauf genommen, gegenüber der Möglichkeit, eine Rechnung mit eventuell - auf Grund eines Fehlers - sinnlosen Zahlenwertes

fortzusetzen. Bei den arithmetischen Unterprogrammen wurde deshalb das CARRY-Bit ausgewählt Fehlermeldungen zu übergeben (beim Rücksprung aus dem Unterprogramm). Dies hat den Vorteil, daß Fehlermeldungen so durch eine Aufrufverschachtelung weitergegeben werden können, bis z.B. im Hauptprogramm durch die Fehlerkennzeichnung ein Fehlerausdruck und ein Programmstopp veranlaßt wird.

Ein Unterprogrammaufruf, z.B. zur Durchführung einer Gleitkommamultiplikation, hat nach den obigen Ausführungen beispielsweise folgende Form:

Aufrufbeispiel Gleitkommamultiplikation:

```
      :  
      :  
LXI   D, OP1   ;   Adresse 1. Operand  
LXI   H, OP2   ;   Adresse 2. Operand  
LXI   B, RES   ;   Adresse Resultat  
CALL  GMUL     ;   UP-Aufruf  
JC    ERROR    ;   Sprung zur Marke ERROR  
                        wenn Fehler  
      :  
      :  
ERROR ;           ;   Fehlerroutine  
      :  
      :
```

Alle weiteren Unterprogramme sind auf die gleiche Art und Weise aufzurufen.

5.1.1 Verwendung der arithmetischen Unterprogramm-bibliothek im "immediate mode".

Zur Überprüfung der Vorgehensweise bei der Programmerstellung sollte eine Konvertierung von Dezimalzahlen in Hexadezimalzahlen und umgekehrt, sowie die Durchführung von "Tischrechnerfunktionen" zu jedem Zeitpunkt der interaktiven Arbeitsweise möglich sein. Während die erste Forderung durch das Kommando CALCULATE bereits erfüllt wird, sind für den zweiten Fall - dem sogenannten *immediate mode* - zusätzliche Systemleistungen notwendig. Der Wunsch zur Einleitung des *immediate modes* wird durch das spezielle Kommando IMMEDIATE angezeigt. Anschließend erfolgt die Interpretierung eingegebener Ausdrücke, wobei die Regeln für die Prioritäten der Operatoren und für die Klammersetzung gleich gewählt wurden

ein Speicherplatz von ungefähr 2k Byte erforderlich, wobei zu berücksichtigen ist, daß diese Unterprogramme in ROM's gespeichert werden können. Die Aufgabe, Unterprogramme mit annehmbaren Rechenzeiten und erforderlicher Genauigkeit für Prozeßaufgaben zu erstellen, konnte durch die angewandte Implementierungsstrategie gelöst werden. Die spezifischen Eigenschaften der implementierten Unterprogramme sind aus der Tab. 2 zu entnehmen.

Unterprogramm	Funktionsbeschreibung	Rechenzeit (ms)	Speicherplatz (Byte)
IA00	Textkomparation	0,00	
IA01	Textkomparation 16 Bit Produkt	1,00	
IA02	Textkomparation 16 Bit Produkt	1,00	
IA03	Zahleneingabe	---	
IA04	Zahlenausgabe	---	
IA05	Textausgabe	---	
* steht für operandenabhängig			

Tab. 2: Spezifische Eigenschaften der implementierten Unterprogramme

Die Angabe der Rechenzeit bezieht sich auf den Prozessor *INTEL 8080A*.

5.2 Prozeßobjekte

Module zur Durchführung von Prozeß Ein-/Ausgabevorgängen werden unter dem Begriff *Prozeßobjekte* zusammengefaßt [58]. Nach [48] hat es sich als günstig erwiesen bestimmte Ein- und Ausgabetransfers zu standardisieren und in der Form sogenannter *Prozeßvariablen* dem Anwender unter einem reservierten Namen zur Verfügung zu stellen.

Da Mikrorechner häufig vom Anwender selbst anhand der zur Verfügung stehenden Halbleiterbausteine erstellt werden, ist der notwendigen Stan-

standardisierung eine Grenze gesetzt. Es erscheint deshalb bei derartigen Kleinstrechnersystemen erforderlich, nur "standardisierte" Prozeßvariablen für die Ein- und Ausgabe digitaler Information gemäß Abschnitt 4.3.2.9 vorzusehen. Im einzelnen ist zu unterscheiden zwischen:

INB(x),	OUTB(x)	Ein-/Ausgabe eines Bits,
INW(x),	OUTW(x)	Ein-/Ausgabe eines 16-Bit Wortes,
IND(x),	OUTD(x)	Ein-/Ausgabe einer vierstelligen BCD-Zahl,
IN\$(x),	OUT\$(x)	Ein-/Ausgabe eines Bitmusters von 16-Bit Breite.

Der Parameter x dient zur Kennzeichnung der jeweiligen Ein-/Ausgangsleitung bzw. der Ein-/Ausgangskanäle [42, 45]. Die Bearbeitung von Analogsignalen erfordert eine zusätzliche hardwaremäßige Integrierung der gewünschten Analogbausteine in das Systemkonzept, wobei die jeweiligen Ein-/Ausgabetransfers unter Verwendung der obigen Prozeßvariablen abgewickelt werden können.

Bei dem betrachteten Mikroprozessor *INTEL 8080* sind zwei Arten der I/O-Behandlung zu unterscheiden, nämlich das Verfahren des *Isolated I/O* sowie die Möglichkeit des *Memory Mapped I/O*. Während für die jeweiligen Ein-/Ausgabetransfers im Fall des "Isolated I/O" spezielle I/O-Befehle bezüglich einer begrenzten Anzahl von I/O-Ports zur Verfügung stehen, kann im Falle des "Memory Mapped I/O" praktisch der ganze zur Verfügung stehende Adreßbereich für Ein-/Ausgabevorgänge genutzt werden. Da beide Verfahren sich durch gewisse Vorteile auszeichnen [12], sollte der Anwender auch in dieser Hinsicht nicht beschränkt sein.

Im Falle des "Isolated I/O" sind prinzipiell 256 Ein- oder Ausgabeports adressierbar, die jeweils eine Verarbeitungsbreite von acht Bit aufweisen. Für die Ein- oder Ausgabe eines 16-Bit Wortes werden somit im einfachen Fall zwei I/O-Ports benötigt, wobei hardwaremäßig sicherzustellen ist, daß nicht vor Abschluß des gesamten Transfervorganges die Ein-/Ausgabe von Teilinformationen erfolgt. Der Aufruf einer Systemvariablen dieses Typs bewirkt, daß für einen Parameter x die Ports $2 \times x$ und $2 \times x + 1$ angesprochen werden (Abb.10.1). Der maximale zulässige Wert des Parameters x beträgt dann 127 [42].

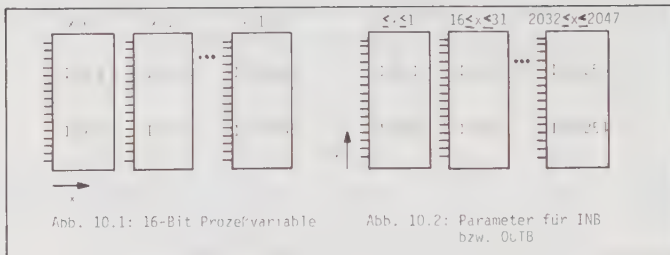


Abb. 10: Parameter für Prozeßvariable

Im Falle der Variablen INB bzw. OUTB kennzeichnet der Parameter x direkt die jeweilige Ein-/Ausgangsleitung. Im Gegensatz zu [45] bezieht sich die Angabe auf 8-Bit breite "Interfaces" (Ports), so daß nach Abb.10.2 bei 256 möglichen Ports der maximale Wert auf $x = 2047$ beschränkt ist.

Um auch die Möglichkeiten des "Memory Mapped I/O" auszuschöpfen, wird für alle Prozeßvariable außer INB bzw. OUTB festgelegt:

Definition 5.1

Bei allen standardisierten Prozeßvariablen mit einer "Verarbeitungsbreite" von 16 Bit wird für Parameterwerte größer als ein zu wählender Grenzwert z die gewünschte Ein-/Ausgabe nach dem Prinzip des "Memory Mapped I/O" durchgeführt. Der Parameter kennzeichnet dann direkt die Adresse des LSB (Least significant byte) der zu bearbeitenden Information.

Für den zu betrachtenden Mikroprozessor *INTEL 8080* wurde aufgrund der Portzahl für den Fall des "Isolated I/O" der Grenzwert $z = 127$ vorgeschlagen.

Aus den obigen Ausführungen ist zu entnehmen, daß der erlaubte Wert eines Parameters x kleiner als 65 335 betragen muß und somit in zwei

Byte abgelegt werden kann. Die jeweiligen Unterprogramme erwarten deshalb als Parameter eine zwei Byte lange Zeichenfolge. Ausdrücke vom Typ <real> als Parameter erfordern eine vorherige Umwandlung.

Für die Übergabe des Parameters an das jeweilige Unterprogramm wurde die in Abschnitt 5.1.3 vorgestellte Form beibehalten, d.h. der Inhalt des Registerpaares DE gibt die Adresse des Parameters, der Inhalt des Registerpaares BC die Adresse der Ein-/Ausgabeinformation an.

5.3 Real-Time Systemvariable

Die Behandlung zeitabhängiger Aufträge erfordert das Vorhandensein einer Realzeit-Uhr. Eine einfache Möglichkeit zu deren Realisation unter Verwendung gängiger Halbleiterbausteine wird in [42] vorgeschlagen. Diese Realzeit-Uhr gestattet die Abfrage der reservierten Systemvariablen

MSEC:	Absolutzeit in Millisekunden	(0...999),
SEC:	Absolutzeit in Sekunden	(0...59),
MIN:	Absolutzeit in Minuten	(0...59) und
HOURL:	Absolutzeit in Stunden	(0...23).

Die Zeitvariable MSEC besitzt eine Auflösung von 1ms. Die Absolutzeit wird unter Verwendung des Kommandos TIME gesetzt bzw. abgefragt.

Der Programmierer kann diese Systemvariablen wie alle anderen Real-Variablen an jeder gewünschten Stelle seines Programms einsetzen. Das Ergebnis eines Aufrufs ist jeweils eine Gleitkommazahl.

6. EIN INTERAKTIVER ÜBERSETZER FÜR MIKRORECHNER

Obwohl die Leistungsfähigkeit der verschiedenen Mikroprozessoren fortwährend verbessert wird, sind z.Zt. noch gewisse Beschränkungen besonders auch hinsichtlich Verarbeitungsbreite und Rechenzeit nicht zu umgehen. Derartige Beschränkungen müssen bei der Konzeption eines Übersetzers für Mikrorechner berücksichtigt werden und erfordern entsprechende unkonventionelle Verfahren.

6.1 Struktur des Übersetzers

Das gesamte Übersetzersystem ist voraussetzungsgemäß speicherresident, d.h. es steht kein zusätzlicher Hintergrundspeicher - z.B. Magnetbandkassette oder Floppy Disk - zur Verfügung. Aus Gründen der Informationssicherung sollte somit der Übersetzer selbst in einem nichtflüchtigen Speichermedium (ROM) abgelegt werden, während zur Speicherung der temporären Informationen zusätzlicher Speicherplatz - üblicherweise mit RAM bezeichnet - erforderlich ist. Bei der Realisation des Übersetzerkonzeptes war deshalb darauf zu achten, daß die Vorgehensweise bei der Programmierung der jeweiligen Module deren Speicherung im ROM-Bereich zuließ.

Interaktive Softwareerstellung zeichnet sich u.a. dadurch aus, daß der Programmierer jeweils sofort nach Eingabe einer Zeile über eventuelle Programmierfehler informiert wird. Die Eingabeinformation ist also in einer sogenannten *Analysephase* zu prüfen. Bei korrekter Eingabe kann nun entweder der fehlerfreie Quellencode selbst zur weiteren Bearbeitung intern abgelegt oder der nahezu endgültige Maschinencode bereits schon in dieser Phase erzeugt werden (inkrementeller Übersetzer [92]). Bei der zuletzt genannten Vorgehensweise ist es allerdings dann nicht mehr möglich, den eingegebenen Quellencode zurückzugewinnen, sofern nicht eine zusätzliche Quellenkopie intern abgelegt wird. Außerdem erfordert eine derartige Arbeitsweise einen relativ großen Organisationsaufwand [58]. Für das zu diskutierende Übersetzerkonzept wurde deshalb ein Verfahren gewählt, welches der Methode der Speicherung des

fehlerfreien Quellencodes ähnlich ist, aber mehr den Erfordernissen der Mikrorechnersysteme entspricht. Statt des Quellencodes selbst wird ein kompakter *Zwischencode* gespeichert, aus dem sich zu jeder Zeit das ursprüngliche Quellenprogramm zurückgewinnen läßt.

Aufgrund der zeilenweisen Syntaxprüfung ist dieser Zwischencode abgesehen von eventuellen Fehlern bezüglich der Gesamtstruktur eines Programms (z.B. FOR-Schleifen ohne NEXT, Sprünge ohne Sprungziel) fehlerfrei. Läßt man noch zusätzlich für diese Phase der Übersetzung die selteneren *mixed mode* Fehler zu, so kann ein relativ einfaches Verfahren zur Durchführung der Syntaxprüfung realisiert werden. Fehlerhafte Zwischencodezeilen der oben beschriebenen Art sind auf einfachem Wege zu Beginn der *Codegenerierungsphase* zu erkennen. Somit gilt:

Definition 6.1

Der Zwischencode ist eine kompakte, weitgehend fehlerfreie systeminterne Form des Quellenprogramms.

Der Abschluß der Analysenphase wird durch Eingabe des Kommandos COMPILER gekennzeichnet, welches nun den Start der Generierungsphase initialisiert. Außer der Kennzeichnung eventuell noch vorhandener Fehler und der Durchführung einiger Vorbereitungen zur eigentlichen Codegenerierung, z.B. Berechnung der Basisadresse für Variable, erfolgt nun die Übersetzung des vorliegenden Zwischencodes in den Maschinencode des jeweiligen Systems.

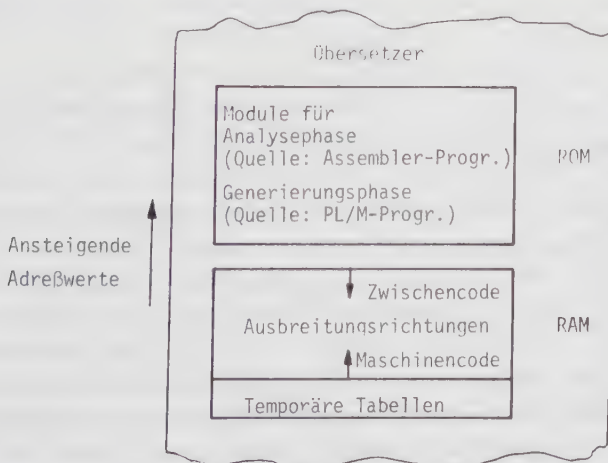


Abb.11: Prinzipielle Aufteilung eines Adreßbereiches für Module und Tabellen des Übersetzers

Während der Analysephase müssen neben dem eigentlichen Zwischencode auch eine Anzahl von Tabellen zur Abwicklung der Analyse sowie für die folgende Generierungsphase angelegt werden. Zur Vereinfachung der Tabellenverwaltung wurde in dem Übersetzerkonzept festgelegt, daß die gleichzeitig wachsenden Bereiche für Tabellen und Zwischencode - wie in Abb. 11 angedeutet - sich in entgegengesetzter Richtung ausbreiten.

Die Organisation von Tabellen und die Optimierung des Suchprozesses stellt für herkömmliche Übersetzer ein wichtiges Problem dar, da hiervon ebenfalls die benötigte CPU-Zeit des Übersetzungsvorganges abhängt. Bei der vorgegebenen Aufgabe kann dem Punkt des Rechenzeitbedarfes der Übersetzung ein geringer Stellenwert beigemessen werden, da das System ausschließlich für Übersetzungsaufgaben zur Verfügung steht und somit keine anderen Programmläufe blockiert. Deshalb war es möglich, die Tabellen für lineares Suchen zu organisieren. Trotz dieser Maßnahme wird die Analyse einer Zeile so schnell abgewickelt, daß der Programmierer davon kaum etwas bemerkt. Der Zeitbedarf für die eigentliche Codegenerierung beträgt auch bei umfangreichen Programmen - wie bei verschiedenen Tests festzustellen war - nur einige Sekunden, so daß die getroffene Entscheidung bestätigt wurde.

Die Frage nach der Wahl der Programmiersprache zur Erstellung der Übersetzermodule selbst ist eine der Fragen, die am Anfang des Konzeptentwurfes beantwortet werden muß. Wie bereits erläutert, lassen sich Softwaremodule für Mikroprozessoren bei Einsatz entsprechender Cross-Software auf einer größeren DV-Anlage sowohl in einer Assemblersprache als auch in einer höheren Programmiersprache erstellen. Für das betrachtete System INTEL 8080 war eine Auswahl zwischen der Assemblersprache und der höheren Programmiersprache PL/M zu treffen.

Die eigentliche Analysephase des erarbeiteten Übersetzerkonzeptes ist systemunabhängig, da in dieser Phase die Analyse des Quellenprogrammes und die Erzeugung des Zwischencodes erfolgt. Häufige Änderungen dieser Programmstruktur sind deshalb auszuschließen. Um möglichst kompakte, rechenzeiteffektive Module zu erstellen, wurden diese Module in der entsprechenden Assemblersprache geschrieben.

Nach Def. 6.1 stellt der Zwischencode eine kompakte, zeilenweise syntaxgeprüfte Form des Quellenprogramms dar. Dieser systemunabhängige Code wird nun in der sich anschließenden Generierungsphase in den gewünschten Maschinencode übersetzt. Um flexibler hinsichtlich Änderungen in der Struktur des Zielrechners zu sein, bei dem die erstellte Software letztlich zu implementieren ist, wurden die Module für die Generierungsphase in der höheren Programmiersprache PL/M geschrieben (Abb. 11). Dem Nachteil des größeren Speicherbedarfes steht der wesentliche Vorteil der Selbstdokumentation dieser Programme und somit die Möglichkeit gegenüber, Änderungen schnell durchführen zu können. Durch diese Maßnahme erscheint eine einfache Anpassung an veränderte Strukturen des Zielrechners möglich.

6.2 Lexikalische und syntaktische Analyse einer Quellenzeile

Die Analysephase stellt - abgesehen von dem Kommandosystem - den dialogfähigen Teil des Konzeptes dar. Ein wichtiges Modul dieses Übersetzerteiles ist die sogenannte *Eingabephase*, auf welche die folgenden lexikalischen und syntaktischen Analysevorgänge aufbauen.

6.2.1 Die Eingabephase

Die Eingabephase hat die Aufgabe, Zeichen von einem Eingabegerät zu lesen und zu klassifizieren (Typbestimmung). Durch diese Maßnahme wird eine einfache Organisation des Analysevorganges ermöglicht. Bei der Klassifizierung ist zu unterscheiden zwischen Zeichen, die für RT BASIC signifikant sind und speziellen Steuerzeichen. Diese Steuerzeichen wurden deshalb in das Konzept übernommen, um dem Programmierer eine beschränkte Beeinflussung des Eingabevorgangs zu jedem beliebigen Zeitpunkt ohne Einsatz eines umfangreichen Kommandosystems zu ermöglichen.

Die gewählte Klassifizierung der verschiedenen Eingabezeichen ist aus Tab. 3 ersichtlich. Es müssen sich die Steuerzeichen möglichst einfach von den anderen Elementen unterscheiden. Deshalb wird bei diesen Zeichentypen das höchstwertigste Bit gesetzt. Bei der Klassifizierung der Son-

Eingabezeichen	ASCII-Code (Hexa-Zahl)	Typbyte (Hexa-Zahl)
<i>Steuerzeichen</i>		
@ Fluchtsymbol: Kommandoleitung	40	80
CR Wagenrücklauf: Start Syntaxprüfung	0D	81
^ Accent circonflexe: Zeile löschen	5E	82
\ Letztes Zeichen löschen	5C	83
EM End of Medium: Ende Übersetzertätigkeit	19	84
STX Start of Text: Eingabe von LSL	02	85
ETX End of Text: Eingabe von Konsole	03	86
DC1 Ein-/Ausschaltung der Protokollierung	11	87
CAN Normierung des Übersetzers	18	88
<i>Sonderzeichen</i>		
>	3E	01
<	3C	02
*	2A	03
/	2F	04
†	23	05
%	25	06
=	3D	07
-	2D	08
+	2B	09
,	2C	0A
(	28	0B
)	29	0C
.	2E	0D
"	22	0E
Weitere Sonderzeichen außer \$		0F
Ziffern	30 - 39	10
Buchstaben A - Z sowie \$	41 - 5A, 24	20
Leerzeichen	20	40
Restliche Codeelemente (nichtsignifikante Zeichen)		00

Tab. 3: Klassifizierung der Eingabezeichen

der Zeichen wurde besonders berücksichtigt, daß diese Operatoren später ohne größere Modifikationen in die Operatorentabelle eingeordnet werden können. Die restlichen Zeichenklassen, d.h. die Buchstaben, die Ziffern und das Leerzeichen, sind in dem entsprechenden Typbyte bereits durch Abfrage eines Bits zu erkennen. Sonderzeichen, die keine spezielle Bedeutung in der gewählten Programmiersprache haben, werden zu der Klasse OFH zusammengefaßt. Die restlichen Zeichen erhalten die Klassifizierung OOH und sind bei der Eingabe als nichtsignifikant zu überlesen. Durch diese Maßnahmen vereinfachen sich die bei Analyse- und Suchroutinen oft nötigen Abfragen, ob ein Buchstabe, eine Ziffer oder ein alphanumerisches Zeichen vorliegt, auf einfache AND-Verknüpfungen.

Ein vereinfachtes Konzept zur Abwicklung der Eingabephase ist aus Abb. 12 ersichtlich. Nach Übernahme des Zeichens wird anhand des entsprechenden ASCII-Wert über eine feste Tabelle das jeweilige Typbyte bestimmt. Falls es sich um ein nichtsignifikantes Zeichen handelt, ist dieses Zeichen zu überlesen. Alle anderen Eingabezeichen werden - sofern es sich nicht um Steuerzeichen handelt - in einem speziellen Eingabepuffer (ZCHNP) zur weiteren Bearbeitung abgelegt. Die Speicherung des jeweiligen Zeichentyps erfolgt in einem Typenpuffer (ZTYPP). Beide Puffer sind so konzipiert, daß das erste Byte eines Puffers die abgelegte Zeichenzahl angibt. Diese Konzeption der Zwischenpufferung erlaubt es neben der eigentlichen Vorbereitung der Analysephase elementare Textmanipulationen durchzuführen. So kann auf einfache Art und Weise mit Hilfe der Steuerzeichen "\ " oder "^ " entsprechend Tab. 3 durch Manipulation der Puffer entweder die gerade bearbeitete Zeile oder das zuletzt eingegebene Zeichen gelöscht werden. Die Durchführung der Zeichenklassifizierung einer Quellenzeile zeigt zusammenfassend Abb. 13 am Beispiel einer LET-Anweisung. Die Analyse einer Zeilenstruktur ist nun unter Verwendung der zusätzlichen Typ-Information organisatorisch wesentlich einfacher abzuwickeln.

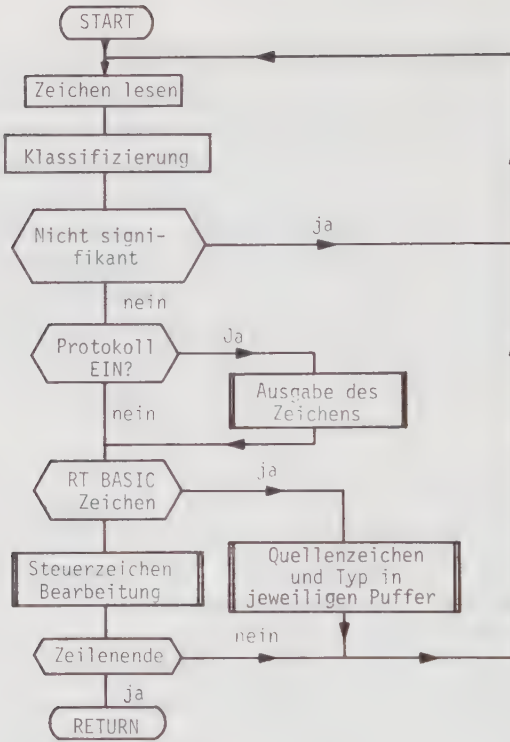


Abb. 12: Struktur des Eingabeprogrammes

CHARAKTER																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																				
-----------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

6.2.2 Tabellenorganisation

Jede fehlerfreie Quellenzeile repräsentiert einen oder mehrere entsprechend der Syntaxdefinition gültigen Sätze der gewählten Programmiersprache. Im Falle der Programmiersprache BASIC bzw. einer RT BASIC-Erweiterung, stellen diese Sätze eine syntaxgerechte Synthese der folgenden Satzelemente dar:

- Zeilenmarkierungen (Labels),
- Schlüsselwörter zu Beginn eines Statements, z.B. LET, GOTO, ..., hier Anweisungsnamen genannt,
- Schlüsselwörter innerhalb eines Statements, z.B. THEN, TO, STEP, ..., hier als "reservierte Wörter" bezeichnet,
- Operatoren, auch Wortoperatoren (MIN, MAX, NOT, ...),
- Sprungziele,
- Konstanten, Variablen und Standardunterprogramme.

Um die Übersetzung eines Quellenprogramms durchzuführen, benötigt der Übersetzer die jeweils gültigen Satzelemente der Sprache in Form von Tabellen. Während bei der Generierungsphase Zugriffe nur auf vorhandene Tabellenstrukturen erfolgen, sind durch die Analysenphase bestimmte Tabellen erst zu erzeugen bzw. zu vervollständigen.

Entsprechend der Konzeption, ist intern jede Quellenzeile in einer bestimmten Zwischencodeform darzustellen. Eine komprimierte Darstellung der Quelleninformation wird dabei durch eine Folge von Schlüsselwörtern der jeweiligen Tabellen repräsentiert. Wegen der Bytestruktur des betrachteten Mikroprozessors, sollte die Verarbeitungsbreite eines Schlüsselwortes aus Gründen der Rechenzeit und des Speicherbedarfes ebenfalls nur ein Byte betragen. In diesem Byte ist dann neben der Art des Tabelleneintrages, zusätzlich die Stellung des jeweiligen Satzelementes in der Tabelle zu codieren.

Berücksichtigt man die Zeilenmarkierungen vorläufig nicht, so lassen sich die angeführten Satzelemente gemäß Tab. 4 zu Gruppen zusammenfassen. Entsprechend der Anzahl der vorzunehmenden bzw. zu erwartenden Eintragungen in die jeweilige Tabelle, kann nun eine Codierung für die zu-

zuordnenden Schlüsselwörter - wie in Tab.4 angedeutet - angegeben werden. Durch diese Konzeption ist eine je nach Tabellenart unterschiedliche

Gruppe	Schlüsselwörter	Bemerkungen
Symbole	Oxxx xxxx	Hinweis auf alle Variablen, Konstanten und Standardunterprogramme
Sprungziele	LLxx xxxx	
Anweisungsnamen	LOOx xxxx	
Operatoren	LOLx xxxx	
Reservierte Wörter	LOOx xxxx	Gleicher Codierungsbereich wie bei den Anweisungsnamen.

Tab. 4: Codierung der Tabelleneinträge

obere Grenze für die Anzahl der Eintragungen vorgegeben, die jedoch auch bei umfangreicheren Programmen nur selten zu erreichen sein wird. In der bisherigen Testphase machte sich eine derartige Begrenzung nicht bemerkbar.

Die Symboltabelle

In der Symboltabelle sind neben den unabhängig vom Quellprogramm vermerkten Standardfunktionen und Systemprozeduren, in Abhängigkeit von der Analysephase Angaben zu Konstanten, Variablen, Feldern, Strings, usw. gespeichert.

Jede Eintragung besteht aus einem sogenannten Längenbyte, dem Namen des einzutragenden Elementes in ASCII-Codierung, dem Schlüssel der Eintragung nach Tab.4 , einem Beschreibungsbyte, in welchem ergänzende Informationen zu dem jeweiligen Eintrag vermerkt sind, Speicherplatz für die später festzulegende Basisadresse des Elementes sowie eventuell einem oder

Längen-	Namen	Schlüssel	Beschrei-	Basisadresse	Indexwerte
	...				...
byte	max. 6 Zeichen		bund		

Abb. 14: Eintrag in die Symboltabelle

mehrerer Indexparameter. Das Format eines derartigen Eintrages in die Symboltabelle zeigt Abb. 14.

Da das Format der Symboltabelleintragen eine variable Länge aufweist, erscheint es zweckmäßig, am Anfang einer Eintragung deren Länge in Form einer Dualzahl zu vermerken. Handelt es sich bei dem Eintrag um eine Konstante, so wird an die entsprechende Stelle ein intern erzeugter Name, der für alle Konstanten identisch ist, nämlich die ASCII-Codierung für das Zeichen "*" eingetragen.

Die vorgestellte Konzeption macht - wie in Abb.14 angedeutet - die Speicherung einer Symbolbeschreibung nach Abb.15 erforderlich, da in der Symboltabelle Eintragungen unterschiedlichen Typs abgelegt werden. Diese

Indexwert	Nummern des Beschreibungsbytes					
	0	1	2	3	4	5
Konstante	0	0	0: arithm.			L: Hexalok. Konstante
Variable						
Label						
IP		1	0: Standard-Fkt. 1: Syst.-Proc.	0: Input 1: Output		Parameterzahl

Abb. 15: Beschreibungsbyte eines Symbols

Symbolbeschreibung enthält alle spezifischen Kennzeichen eines Eintrags, die in der weiteren Analyse - bzw. Generierungsphase von Bedeutung sind.

Die Belegung der, nach der Beschreibung folgenden Adresse, wird erst zu Beginn der Generierungsphase durchgeführt. Diese Adresse zeigt bei arithmetischen Konstanten und Variablen auf das LSB der Information, bei Feldern auf das LSB des ersten Feldelementes und bei Strings auf das Längenbyte der Stringinformation.

Die Konstantentabelle

Bei der Bearbeitung einer Konstanten während der Analysephase erfolgt - sofern es sich um eine bisher noch nicht bekannte Konstante handelt - nach der Speicherung der entsprechenden Merkmale in der Symboltabelle der Eintrag des eigentlichen Konstantenwertes in die Konstantentabelle. Dabei besteht jeder Eintrag aus einem Längenbyte, gefolgt von der Information. Während arithmetische Konstanten als drei Byte Gleitkommazahlen abgelegt werden, besteht die Information bei Stringkonstanten aus einer Folge von ASCII-Zeichen bzw. aus einer Hexazahl.

Tabelle der Anweisungsnamen

Über diese "feste" Tabelle wird für jeden im Rahmen der Sprachdefinition zulässigen Anweisungsnamen ein Schlüssel entsprechend Tab.4 zugeordnet. Jede Eintragung der Tabelle besteht aus dem Anweisungsnamen im ASCII-Code, gefolgt von dem jeweiligen Schlüssel. Die drei ersten Bits eines Schlüssels beinhalten als Hinweis auf den Anweisungsnamen das Bitmuster 100, die fünf restlichen Bits bilden die Nummer der Anweisung in der Tabelle, so daß 32 verschiedene Anweisungsnamen zu unterscheiden sind.

Tabellen der Operatoren und reservierten Wörter

Der Aufbau dieser ebenfalls nicht wachsenden Tabellen ist prinzipiell identisch mit der Tabelle der Anweisungsnamen.

Der Schlüssel der Operatoren beginnt mit der Bitkombination LOL, wiederum gefolgt von fünf Bits, welche die Stellung des Operators in der Tabelle kennzeichnen. Somit können ebenfalls 32 Eintragungen verschlüsselt werden.

Die Schlüssel für die reservierten Wörter benutzen den gleichen Wertebereich wie die der Anweisungsnamen. Durch die unterschiedliche Stellung dieser Satzelemente innerhalb einer Zeile ist jedoch eine Verwechslung ausgeschlossen.

Sprungziele

Anweisungen, in denen Marken als Sprungziele vorkommen, z.B. nach einer GOTO-Anweisung, initialisieren eine Eintragung in die Sprungzieltabelle. Für eine derartige Eintragung wurde das folgende Format vereinbart:

An der ersten Stelle steht jeweils ein sogenanntes "Statusbyte", gefolgt von einem zwei Byte umfassenden Markenfeld, in welches das Quellensprungziel als 16 Bit Festkommazahl einzutragen ist. Daran schließt sich das ebenfalls zwei Byte große Adreßfeld an, in das bei Durchführung der Generierungsphase dann die endgültige Objektcodeadresse abgelegt wird.

In dem Statusbyte werden während der Analysephase bzw. zu Beginn der Generierungsphase spezifische Informationen über dieses Sprungziel gespeichert, die dann zum gewünschten Zeitpunkt zur weiteren Bearbeitung des jeweiligen Sprungzieles zur Verfügung stehen. So dient z.B. Bit 0 dieses Statusbytes zur Unterscheidung, ob der Inhalt des Adreßfeldes schon das Sprungziel selbst oder noch eine Verknüpfungsadresse ist (bei Bearbeitung von Vorwärtssprüngen).

Der Schlüssel für Sprungziele beginnt nach Tab. 4 mit dem Bitmuster LL, gefolgt von einer sechs Bit langen Nummer des Tabellenplatzes. Es können somit maximal 64 verschiedene Sprungziele bearbeitet werden.

Anordnung der Tabellen

Von den besprochenen Tabellen wird die Größe der Symboltabelle, der Konstantentabelle und der Sprungzieltabelle von dem Umfang des eingegebenen

Quellenprogrammes bestimmt. Da der Speicherbedarf dieser drei Tabellen somit nicht exakt vorhersehbar ist, muß eine Verschiebung der restlichen Tabellen nach Ausschöpfung des jeweils vorgegebenen Speicherbereiches einer Tabelle gewährleistet sein. Da die Symboltabelle bei der Bearbeitung eines Programmes von den temporären Tabellen den größten Speicherbereich beansprucht, wird sie an die erste Stelle des temporären Tabellenbereiches angelegt, so daß - sofern erforderlich - nur nachfolgende Tabellen zu verschieben sind. Die Konzeption zur Verschiebung der jeweiligen Tabellen ist in [25] dargestellt. Bei Implementierung der DATA-Anweisung muß noch die zusätzlich anzulegende DATA-Tabelle in den verschiedenen Verschiebevorgang einbezogen werden.

6.2.3 Interaktive Analyse

Nach SCHNEIDER [92] dient die syntaktische Analyse zur Prüfung des "korrekten" Aufbaues eines Quellenprogrammes. Dabei kennt man verschiedene Möglichkeiten, die Struktur des Programms für folgende Arbeitsphasen festzuhalten, nämlich

- die Generierung eines Strukturbaumes,
- die Umformung des Programms in eine "polnische Notation" und
- die Generierung von Makrobefehlen.

Bei dem vorgegebenen Übersetzerkonzept soll jedoch der auf Grund der lexikalischen Aufbereitung des eingegebenen Quellenprogrammes und der sich anschließenden Syntaxprüfung erzeugte Zwischencode, nicht nur als Grundlage der sich anschließenden Generierungsphase dienen, sondern zusätzlich die Rekonstruktion des Quellenprogramms gestatten. Für diese Übersetzerphase wurde deshalb nicht auf eines der in der Literatur [z.B.66, 74, 88, 92] beschriebenen "klassischen" Verfahren zurückgegriffen, sondern ein einfaches und schnelles Analyseverfahren zur weitgehenden Syntaxprüfung (siehe auch Abschnitt 6.1) einer Zeile vorgeschlagen und implementiert.

Die im folgenden zu beschreibende Analysephase ist eine Kombination aus lexikalischer und syntaktischer Analyse. Die lexikalische Analyse hat wie üblich die Aufgabe, die von der Eingabephase übergebenen Einzelzeichen mit Hilfe von Tabellen zu syntaktischen Einheiten zusammenzufassen und wenn er-

forderlich, die entsprechenden Tabellen fortzuschreiben. Auch die Umschlüsselung von Doppeloperatoren und richtige Identifizierung kontextabhängiger Operatoren (z.B. für "=" Kennzeichnung ob Zuweisungs- oder Relativoperator) fällt unter diese Aufgabe [25].

Die Durchführung der syntaktischen Analyse einer Quellenzeile wird durch das nachfolgend dargestellte Verfahren abgewickelt:

Jede Anweisung besitzt eine, durch die Syntax der Anweisung festgelegte Struktur, welche anhand der Schlüsselwörter einfach zu erkennen ist. Eine wichtige Aufgabe besteht deshalb in der Feststellung, ob die in der Syntax vorgegebene "Rahmenstruktur" einer Anweisung eingehalten wurde. So erfolgt nach der Kontrolle der Zeilennummer in Abhängigkeit von dem Anweisungsnamen eine Verzweigung in spezielle Programme, die jeweils eine weitgehende Analyse gestatten. Nach der Identifizierung einer syntaktischen Einheit im Rahmen der lexikalischen Analyse, schließt sich jeweils eine sofortige Prüfung hinsichtlich der Stellung dieses syntaktischen Elementes innerhalb der Eingabezeile an. Ergibt sich keine Beanstandung, so wird das entsprechende syntaktische Element als ein Element des Zwischencodes in den Zwischencodepuffer abgelegt.

Analyse von Ausdrücken

Die Syntaxanalyse von Ausdrücken ist eine der wichtigsten Aufgaben im Rahmen der Analysephase. Das zu diesem Zweck angewandte Konzept soll nachfolgend diskutiert werden.

Ein Ausdruck ist immer dann als fehlerhaft anzusehen, wenn die Anordnung eines Ausdruckelementes nicht der Syntax entspricht oder ein Fehler in der Klammerstruktur vorliegt (z.B. mehr öffnende als schließende Klammern). Im Falle eines Anweisungsausdruckes darf - wie aus der Syntax zu entnehmen ist - zusätzlich nur eine Zuweisung erfolgen. Zur Untersuchung der korrekten Stellung eines Elementes innerhalb eines Ausdruckes dient die folgenden Betrachtungsweise:

Entsprechend der Syntax besteht jeder Ausdruck aus einer begrenzten Zahl möglicher Arten von Satzelementen X^i . Die Menge M_A aller möglichen Elementen

te X^i eines Ausdruckes sei

$$(6) \quad M_A = \{X^1, X^2, \dots, X^i, \dots, X^n\},$$

$n \triangleq$ Zahl der maximal möglichen Arten der Satzelemente.

Der Menge M_{X^i} sollen diejenigen Elemente angehören, auf die das Element X^i syntaktisch korrekt folgen darf.

$$(7) \quad S : X^i \rightarrow M_{X^i}, \quad M_{X^i} \subset M_A$$

Die zugehörige Abbildungsvorschrift wird durch die Syntax festgelegt.

$$(8) \quad M_{X^i} = \{X_1^{l_1}, \dots, X_i^{l_i}, \dots, X_p^{l_p}\}, \quad \begin{matrix} l_i \in [1:n] \\ 0 < p \leq n \end{matrix}$$

Ein vorliegender Ausdruck

$$(9) \quad \alpha_A = X_1^{j_1} X_2^{j_2} X_3^{j_3} \dots X_m^{j_m}, \quad j_i \in [1:n]$$

$m \triangleq$ Zahl der Satzelemente

ist dann hinsichtlich der Anordnung der Elemente syntaktisch korrekt, sofern gilt:

$$(10) \quad X_q^{j_q} \in M_{X_{q+1}^{j_{q+1}}}, \quad \begin{matrix} q \in [1:m-1] \\ j_q \in [1:n] \end{matrix}$$

Die Satzelemente X^i sollten in einer möglichst einfachen Form für die Durchführung der obigen Abfrage im Rahmen einer Implementierung gekennzeichnet werden. Die gewählte "Bezeichnung" (abgekürzt $BEZ(X^i)$) sei:

$$(11) \quad BEZ(X^i) := s_u^i s_{u-1}^i \dots s_1^i$$

$$s_k^i := \begin{cases} 1 & \forall k=i \\ 0 & \forall k \neq i \end{cases}, \quad \begin{matrix} k \in [1:u] \\ u \geq n \\ i \in [1:n] \end{matrix}$$

Bestimmt man weiterhin für die "Bezeichnung" der Menge M_{X^i}

$$(12) \quad \text{BEZ}(M_{X^i}) := t_u^i t_{u-1}^i \dots t_1^i$$

$$t_r^i := \begin{cases} 1, & \text{wenn } X^r \in M_{X^i} \forall r \in [1:u] \\ \text{sonst } 0 \end{cases}$$

$$i \in [1:n] \quad , \quad u \geq n$$

so läßt sich die Abfrage nach Gleichung (10) auf eine einfache logische UND-Verknüpfung der jeweiligen Bezeichnungen zurückführen. Eine Folge von Satzelementen ist demnach syntaktisch zulässig, sofern die Bedingung

$$(13) \quad Y_q = \text{BEZ}(X_q^{j_q}) \wedge \text{BEZ}(M_{X_{q+1}^{j_{q+1}}})$$

$$Y_q \neq 0, \quad \begin{cases} q \in [1:m-1] \\ j_q \in [1:n] \end{cases}$$

erfüllt ist.

Die geschilderte Verfahrensweise sei nun an dem Beispiel der vorliegenden Syntax von Ausdrücken (Abschnitt 4.3.3) gezeigt.

Es gilt:

$$M_A = \{X|X^1 = \text{OPD}, X^2 = \text{OPT}, X^3 = \text{UOPT}, X^4 = \text{OK}, X^5 = \text{SK}, X^6 = \text{ZW}\}$$

$$n = 6.$$

Dabei steht die Abkürzung OPD für Operand, OPT für binärer Operator, UOPT für unärer Operator, OK für öffnende Klammer, SK für schließende Klammer und ZW für Zuweisung. Aus der Syntax ist zu entnehmen:

$$M_{X^1} = \{\text{OPT}, \text{UOPT}, \text{OK}, \text{ZW}\},$$

$$M_{X^2} = \{\text{OPD}, \text{SK}\},$$

$$M_{X^3} = \{\text{OPT}, \text{UOPT}, \text{OK}, \text{ZW}\},$$

$$M_{X^4} = \{OPT, UOPT, OK, ZW\},$$

$$M_{X^5} = \{OPD, SK\} \text{ und}$$

$$M_{X^6} = \{OPD, SK\}$$

Da nur sechs verschiedene Arten von Satzelementen zu verzeichnen sind, kann die Bezeichnung von X^i in einer Speichereinheit des betrachteten Systems (1 Byte = 8 Bit) abgelegt werden ($u = 8$).

$$BEZ(X^i) = s_u^i s_{u-1}^i \dots s_1^i, \quad u = 8, n = 6$$

$$S_1^1 = 1, S_2^1 = 0, \dots, S_8^1 = 0 \Rightarrow BEZ(X^1) = 0000\ 0001 = 01H$$

$$S_1^2 = 0, S_2^2 = 1, \dots, S_8^2 = 0 \Rightarrow BEZ(X^2) = 0000\ 0010 = 02H$$

$$BEZ(X^3) = 04H, BEZ(X^4) = 08H, BEZ(X^5) = 10H, BEZ(X^6) = 20H$$

$$BEZ(M_{X^i}) = t_u^i t_{u-1}^i \dots t_1^i, \quad u = 8, n = 6$$

$$t_1^1 = 0, t_2^1 = 1, t_3^1 = 1, t_4^1 = 1, t_5^1 = 0, t_6^1 = 1,$$

$$t_7^1 = 0, t_8^1 = 0 \Rightarrow BEZ(M_{X^1}) = 0010\ 1110 = 2EH$$

$$BEZ(M_{X^2}) = 11H, BEZ(M_{X^3}) = 2EH, BEZ(M_{X^4}) = 2EH$$

$$BEZ(M_{X^5}) = 11H, BEZ(M_{X^6}) = 11H.$$

Die jeweiligen Beschreibungen sind in der Form einer festen Tabelle abgelegt. Ein Zugriff wird über die Codierung der gerade bearbeiteten syntaktischen Einheit abgewickelt. Sofern jeweils die Bedingung (13), erfüllt ist und keine Beanstandungen hinsichtlich der Klammerschachtelung und der Rahmenstruktur erfolgen [25], wird die analysierte Quellenzeile als "fehlerfrei" akzeptiert.

Zwischencode

Wie in Definition 6.1 gefordert, soll der Zwischencode als eine kompakte, fehlerfreie und systeminterne Form des Quellenprogramms vorliegen. Aus dem Zwischencode müssen die eingegebenen Quellenzeilen jeweils wieder reproduzierbar sein. Da die syntaktischen Elemente einer Quellenzeile entweder in einer "festen" Tabelle des Übersetzers bereits vermerkt sind oder im Rahmen der Analysephase in einer Tabelle abgelegt wurden, läßt sich eine Quellenzeile prinzipiell auf Grund der vorliegenden Tabelleninformationen reproduzieren.

Der Zwischencode sei bis auf die dual codierte Zeilennummer eine Folge von Tabellenschlüsseln, sofern es sich um ausführbare Anweisungen handelt.

Im Falle der nichtausführbaren Anweisungen besteht die jeweilige Zwischen-codezeile - abgesehen von der Zeilennummer und der Verschlüsselung des Anweisungsnamens - aus einer Folge der ASCII-Zeichen. Zeilennummern werden immer als zwei Byte umfassende Dualzahlen abgelegt.

Beispiel:

Quellenzeile: 20 LET DIF = ENDE - ANF

Schlüssel des Anweisungsnamens "LET": 8EH

Operatoren-Schlüssel:

Operator "=": Schlüssel A7H

Operator "-": Schlüssel A8H

Annahmen hinsichtlich der Variablenschlüssel:

Variable DIF: Schlüssel 18H

Variable ENDE: Schlüssel 10H

Variable ANF: Schlüssel 12H

Die betrachtete Zwischencodezeile hat dann die Form (hexadezimale Darstellung):

00 14 8E 18 A7 10 A8 12
Zeilennummer

Nach der Syntaxanalyse einer Quellenzeile wird - sofern kein Fehler vorliegt - die Einordnung der entsprechenden Zwischencodezeile in den bereits vorliegenden Zwischencodebereich vorgenommen. Würde dabei nicht ein eindeutiges Ordnungsprinzip angewandt (z.B. Einordnung nach der Größe der Zeilennummer), so wären zusätzliche Angaben notwendig - z.B. eine zusätzliche Tabelle - um Programmteile folgenrichtig zuzuordnen.

Definition 6.3

Die Speicherung der Zwischencodezeilen erfolgt geordnet nach ansteigenden Zeilennummern ohne Zwischenraum.

Die interaktive Arbeitsweise erfordert die Möglichkeit, alte Zeilen des Quellenprogramms zu "überschreiben", zu löschen oder zusätzliche Zeilen einzufügen. Das bedeutet jedoch, daß vor Einordnung einer entsprechenden Zwischencodezeile die Verlagerung aller folgenden Zeilen um die Differenz von alter und neuer Zeilenlänge vorangestellt werden muß. Nach dem Ende dieser Organisationsphase schließt sich der Transfer der vorliegenden Zwischencodezeile von dem Zwischencodepuffer zu dem Zielort im Zwischencodebereich an.

Das Ende der Rechenzeit zur Analyse einer Quellenzeile und zur Einordnung der jeweiligen Zwischencodezeile, zeigt das Übersetzersystem durch Ausgabe der Steuerzeichen "Wagenrücklauf" und "Zeilenwechsel" an. Diese Zeitspanne nach Eingabe der Zeilenendmarkierung ist für den Programmierer kaum wahrnehmbar. Darum stellt die oben skizzierte Vorgehensweise der Syntaxanalyse und Zwischencodebehandlung für den Programmierer keine Begrenzung dar, sondern unterstützt die interaktive Programmerstellung.

Fehlerbehandlung

Werden bei der Eingabe einer Zeile Fehler festgestellt, erfolgt die Ausgabe einer Fehlermeldung nach Abschluß der Zeile. Das Zeichen "?" markiert dabei jeweils den vermuteten Fehlerort unterhalb der fehlerhaften Zeile. Nach der Erkennung eines Fehlers wird versucht die Zeile weiter zu analysieren, wobei alle Quellenzeichen bis zum folgenden "Aufsetzpunkt" durch das Zeichen "\" gekennzeichnet werden.

Die Fehlermeldung sollte die vermuteten Fehler möglichst eindeutig beschreiben, entweder durch Ausgabe von Fehlerstrings oder in Form von Fehlercodes. Die Ausgabe von Fehlerstrings stellt für den Programmierer eine größere Unterstützung dar, erfordert aber auch systemintern zusätzlichen Speicherplatz. Da der vermutete Fehlerort ebenfalls markiert wird, müßte in vielen Fällen allein aus dieser Information die Fehlerart sofort zu erkennen sein, so daß durch Ausgabe der Fehlerstrings - besonders bei Verwendung der "langsamen" Standard-Fernschreibmaschine nach Definition 3.1 - in diesen Fällen für den Programmierer eine lästige Zeitspanne entsteht.

Bei dem betrachteten Übersetzer besteht deshalb aus den oben angeführten Gründen die Ausgabe der Fehlermeldung - zusätzlich zu der Markierung der Fehler - aus einer Folge von Fehlercodes nach Anlage A1.

Eine fehlerhafte Zeile wird nicht in den Zwischencode übernommen und kann nach Ausgabe der Fehlermeldung von dem Programmierer korrigiert wieder eingegeben werden. Das Konzept der Analysephase gestattet somit eine interaktive Programmerstellung. Die Eingabe eines Kommandos (Kapitel 9) ist in jeder Zeile erlaubt.

6.3 Codegenerierung

6.3.1 *Vorbemerkungen*

Das Übersetzerkonzept berücksichtigt die Übersetzung vorliegender Zwischencodebereiche im Rahmen der sogenannten Generierungsphase, welche durch das Kommando COMPILE gestartet wird. Nach dem Ende der Codegenerierung steht dem

Anwender die Möglichkeit offen, das erstellte Objektprogramm sofort zu testen und sofern erforderlich, zu korrigieren. Deshalb darf der Zwischencode nicht zwischenzeitlich durch den zu erzeugenden Maschinencode zerstört werden (siehe auch Abb. 11).

Nach Abschluß der Eingabe eines Programmes erfolgt - initialisiert durch das Kommando COMPILE - die Zuordnung von Speicherplätzen an Variable und Felder, da in dieser Phase der Zwischencode und die temporären Analysetabellen sich nicht mehr verändern. So wird nun auch die Konstantentabelle nicht mehr verschoben und es ist der Eintrag der Konstantenadressen in die Symboltabelle möglich. Diese vorbereitenden Arbeiten werden noch durch weitere Maßnahmen ähnlicher Art ergänzt, bevor die eigentliche Codegenerierung abläuft.

Bei der Abarbeitung des Zwischencodes zum Zwecke der Codegenerierung wird anhand des Schlüsselwortes der jeweiligen Zeile, eine Verzweigung zu entsprechenden Übersetzermodulen durchgeführt, welche dann die Codeerzeugung für diese spezielle Anweisung abwickeln. Ähnlich wie bei der Analysephase, stellt die Codeerzeugung für Ausdrücke eine zentrale Aufgabe der Generierungsphase dar. Das dabei angewandte Konzept soll in dem folgenden Abschnitt betrachtet werden.

6.3.2 Bearbeitung von Ausdrücken

Der auf Grund des einfachen Analyseprinzips gewonnene, bis auf wenige Ausnahmen (siehe Abschnitt 6.1) fehlerfreie Zwischencode eines Ausdrucks, erfordert nun zur Realisierung ein weiterführendes Analyseverfahren. Dabei sollte jedoch die Eigenschaft des "fehlerfreien" Zwischencodes ausgenutzt werden.

Gemäß Abschnitt 4.3.3 gilt für einen Anweisungs-Ausdruck die folgende Syntax, wobei die Zwischensymbole in Großbuchstaben, die Endsymbole in Kleinbuchstaben angegeben sind. Es wurde eine verkürzte Schreibweise gewählt, die im Prinzip aus den Anfangsbuchstaben der einzelnen Symbolbezeichnungen besteht. Für Index- und Parameterklammern erfolgte im Rahmen der vorhergehenden Analysephase eine Umkodierung, so daß sie hier als rechteckige Klammern erscheinen:

```

S    :: =  ⊢AN⊣
AN   :: =  V = E
E    :: =  P1 | E 11 P1
P1   :: =  P2   P1 12 P2
P2   :: =  P3   P2 r1 P3
P3   :: =  P4   P3 r2 P4
P4   :: =  P5 | P4 s P5
P5   :: =  P6   P5 a P6
P6   :: =  P7   P6 m P7
P7   :: =  0 | P7 e 0
0    :: =  u0 | V , F | K | (E)
V    :: =  i{[E{,E}^0_0]}^1_0
F    :: =  i [E]
K    :: =  i

```

Nach R.W. FLOYD liegt immer dann eine Operator-Grammatik vor, wenn die betrachtete Grammatik keine Produktion der Form

$$X \rightarrow \alpha Z_1 Z_2 \beta$$

enthält, wobei X , Z_1 , Z_2 grammatische Klassen (Zwischensymbole) und α , β u.U. leere Satzformen sind. Diese Aussage trifft für die vorliegende Grammatik zu, so daß *Präzedenzrelationen* nur zwischen den Endsymbolen einer Sprache definiert werden brauchen. Da weiterhin die zu bearbeitenden Sätze (Zwischencode) als fehlerfrei anzusehen sind, ist der Einsatz des bewährten *Operator-Präzedenzverfahrens* [69, 74, 76] in Form der wenig Speicherplatz erfordernden *Präzedenzfunktionen* möglich.

Definition 6.4

Zur Kennzeichnung der Präzedenz (Vorrang) zweier benachbarter Operatoren (Begrenzer) werden die in der Literatur üblichen Zeichen verwendet.

- Der linke Begrenzer ist gegenüber dem rechten vorrangig.

- ◁ Gegenüber dem linken Begrenzer hat der rechte Vorrang.
- Beide Begrenzer sind gleichrangig.

Folgen zwei Begrenzer in einer nicht erlaubten Reihenfolge, so soll dies durch "F" gekennzeichnet werden.

links \ rechts		=	l1	l2	r1	r2	s	a	m	e	u	(	)	[	]	,
	≡	<	<	<	<	<	<	<	<	<	<	<	F	<	F	F
=	>	F	<	<	<	<	<	<	<	<	<	<	F	<	F	F
l1	>	F	>	<	<	<	<	<	<	<	<	<	>	<	>	>
l2	>	F	>	>	<	<	<	<	<	<	<	<	>	<	>	>
r1	>	F	>	>	>	<	<	<	<	<	<	<	>	<	>	>
r2	>	F	>	>	>	<	<	<	<	<	<	<	>	<	>	>
s	>	F	>	>	>	>	<	<	<	<	<	<	>	<	>	>
a	>	F	>	>	>	>	>	<	<	<	<	<	>	<	>	>
m	>	F	>	>	>	>	>	>	<	<	<	<	>	<	>	>
l	>	F	>	>	>	>	>	>	>	<	<	<	>	<	>	>
u	>	F	>	>	>	>	>	>	>	<	<	<	>	<	>	>
(	F	F	<	<	<	<	<	<	<	<	<	<	≡	<	F	F
)	>	F	>	>	>	>	>	>	>	>	F	F	>	F	>	>
[	F	F	<	<	<	<	<	<	<	<	<	<	F	<	≡	≡
]	>	>	>	>	>	>	>	>	>	>	F	F	>	F	>	>
,	F	F	<	<	<	<	<	<	<	<	<	<	F	<	≡	≡

Abb. 16: Präzedenzmatrix

Die Präzedenzrelationen zwischen den Begrenzern ergeben sich nach [74, 76] aus den folgenden Regeln.

Es seien X , Y und Z Zwischensymbole, a und b Endsymbole und α , β und δ - u.U. auch leere - Zeichenketten. Dann gilt:

$$(14) \quad a \doteq b, \quad X \rightarrow \alpha ab\beta \text{ oder } X \rightarrow \alpha aYb\beta$$

$$(15) \quad a \triangleleft b, \quad X \rightarrow \alpha aY\beta, \\ Y \rightarrow b\delta \text{ oder } Y \rightarrow Zb\delta.$$

$$(16) \quad a \triangleright b, \quad X \rightarrow \alpha Yb\beta, \\ Y \rightarrow \delta a \text{ oder } Y \rightarrow \delta aZ.$$

Für die angegebene Grammatik läßt sich unter Anwendung der obigen Beziehungen die in Abb. 16 dargestellte Präzedenzmatrix angeben, wobei die Begrenzer $\vdash$ und $\dashv$ durch ein gemeinsames Begrenzersymbol $|$ ersetzt wurden.

Nach FLOYD lassen sich die Vorrangbeziehungen zwischen zwei Begrenzern auch mittels zweier Präzedenzfunktionen f und g ausdrücken, die jedem Begrenzer zwei natürliche Zahlen zuordnen. Es sei S_i der zuletzt bearbeitete und bereits im *Operatoren-Keller (Stack)* abgelegte Begrenzer und S_k der gerade betrachtete Begrenzer. Dann gilt [93]:

$$S_i \doteq S_k \rightarrow f(S_i) = g(S_k)$$

$$S_i \triangleleft S_k \rightarrow f(S_i) < g(S_k)$$

$$S_i \triangleright S_k \rightarrow f(S_i) > g(S_k)$$

Bei der Abarbeitung eines Ausdruckes soll von links nach rechts vorgegangen werden. Der Begrenzer S_i ist somit links von S_k zu finden. Die oben angeführten Definitionen läßt sich auch einprägsamer schreiben, wenn für " f " die Formulierung "links" und für " g " die Bezeichnung "rechts" eingesetzt wird.

$$S_i \doteq S_k \rightarrow \text{links}(S_i) = \text{rechts}(S_k)$$

$$S_i \triangleleft S_k \rightarrow \text{links}(S_i) < \text{rechts}(S_k)$$

$$S_i \triangleright S_k \rightarrow \text{links}(S_i) > \text{rechts}(S_k)$$

Die Vorgehensweise zur Bestimmung der Präzedenzfunktionen anhand der vorher ermittelten Präzedenzmatrix ist z.B. in SCHNEIDER [92] angegeben. Danach ergeben sich als Präzedenzfunktionen für einen Anweisungsausdruck die in Tab. 5

Begrenzer s		=	l1	l2	r1	r2	s	a	m	e	u	(	)	[	]	,
links(s)	1	2	4	6	8	10	12	14	16	18	18	1	18	1	18	1
rechts(s)	1	2	3	5	7	9	11	13	15	17	19	19	1	19	1	1

Tab. 5: Präzedenzfunktionen

angeführten Beziehungen. Es sei nochmals vermerkt, daß der bei Präzedenzfunktionen bestehende Nachteil der schlechteren Erkennung syntaktischer Fehler in dem betrachteten Fall unwesentlich ist, da der Zwischencode bereits fehlerfreier Form vorliegt.

Vorschläge zur Implementierung des Operatorprioritäts-Verfahrens sind in der Literatur angegeben [74, 76, 80, 92, 93]. Die Technik der hier angewandten Implementierung zeigt [32]. Da bei Durchführung der "Verschmelzungsoperation" für einen Operator, die Art der beteiligten Operanden sowieso abzufragen ist, können nun die sogenannten *mixed mode* Fehler ohne zusätzlichen Aufwand erkannt werden.

Um den Umfang des erzeugten Maschinenprogramms möglichst klein zu halten, wird eine *Codeoptimierung* durchgeführt. Im wesentlichen bezieht sich diese Maßnahme auf die Minimierung erforderlicher Ladebefehle vor einem Unterprogrammaufruf, nach dem folgenden Prinzip:

Wie in Kapitel 5 angeführt, sind vor Aufruf eines Unterprogrammes die entsprechenden Registerpaare DE, HL und BC mit den Anfangsadressen der jeweiligen Parameter zu laden. Dies geschieht am einfachsten durch einen sogenannten *load index immediate-* (LXI-) Befehl, der bei dem betrachteten System jedoch einen Speicherplatz von drei Byte erfordert. Ist der Inhalt des entsprechenden Registerpaares auf Grund einer vorherigen Operation ganz bzw. byteweise identisch mit dem neu zu ladenden Adreßwert, so kann auf den prinzipiell erforderlichen LXI-Befehl verzichtet bzw. der andere Teil des Registerpaares durch einen einfachen MOV-Befehl modifiziert werden (ein Byte

Speicherplatz). Diese Vorgehensweise braucht nicht auf die Abarbeitung einer Zeile beschränkt, sondern kann auch "Zeilenüberschreitend" angewandt werden, so lange ein linearer Programmablauf vorliegt (z.B. mehrere LET-Anweisungen). Das Prinzip der Eliminierung bzw. Modifizierung von Ladebefehlen führt zu einer nennenswerten Reduzierung des Maschinencodes [32].

6.3.3 Fehlerbehandlung

Um dem Programmierer eine möglichst schnelle Korrektur restlicher Programmierfehler zu ermöglichen, erfolgt im Rahmen der Generierungsphase bei Registrierung eines Fehlers ein Ausdruck des Fehlercodes nach Anlage A2 und soweit möglich, die Nummer der fehlerhaften Zeile. Im Falle der *mixed mode* Fehler ist eine eindeutige Zuordnung von Fehlerort und Zeilennummer gegeben. Bei Fehlern der Art "FOR ohne NEXT" wird die Zeilennummer der entsprechenden FOR-Anweisung ausgedruckt.

Anders verhält es sich bei Fehlern der Art: "Nicht vorhandenes Sprungziel!" Die jeweiligen Absprungstellen lassen sich dann nur mit einem unverhältnismäßig großem Organisationsaufwand ermitteln (aus dem Zwischencode), so daß dem Programmierer derartige Fehler durch Ausgabe des Fehlercodes und des Sprungzieles angekündigt werden.

Das Übersetzerkonzept ermöglicht die sofortige Korrektur fehlerhafter Zeilen durch einfache Eingabe der Korrekturzeile. Eine interaktive Arbeitsweise ist somit gewährleistet.

7. EIN BINDER FÜR ABSOLUT ADRESSIERTE PROGRAMME

Nach der Übersetzung eines Quellenprogrammes liegt der erzeugte Maschinencode - hier Hauptprogramm genannt - für den betrachteten Mikrorechner *INTEL 8080* in absolut adressierter Form vor. Wurden zur Abwicklung der gestellten Aufgabe Unterprogramme der Unterprogrammbibliothek (Library) benötigt, so sind in dem Hauptprogramm entsprechende Verzweigungen zu den jeweiligen Unterprogrammen der Bibliothek enthalten. Dies bedeutet, daß das erzeugte Maschinenprogramm nur bei den Systemen lauffähig wäre, wo auch die entsprechende Bibliothek implementiert ist.

Eine derartige Vorgehensweise erscheint jedoch für die Programmierung von Mikrorechnern nicht zweckmäßig. Besser geeignet ist das Verfahren des automatischen Anbindens nur derjenigen Unterprogramme, die auch zur Abwicklung der gestellten Aufgabe erforderlich sind. Dieses Verfahren des automatischen Bindens (linkage editing) wird bei Rechnersystemen mit der Möglichkeit der relativen Adressierung allgemein angewandt. In diesem Kapitel soll das Konzept eines Binders für absolut adressierte Programme vorgestellt werden.

7.1 Funktionsgerechte Verschiebung absolut adressierter Programme

Die funktionsgerechte Verschiebung eines Programms in einen anderen Adreßbereich erfordert bekanntlich eine Veränderung der Adreßwerte. Dies bedeutet, daß jedes Element eines Programms daraufhin zu untersuchen ist, ob es modifiziert werden muß oder nicht. Da der Objektcode der betrachteten Mikrorechner nicht explizit unterscheidet zwischen relativen und absoluten Werten, muß diese Information durch eine entsprechende Strategie gewonnen werden.

Ein zu verschiebendes Programm besteht maximal aus drei Blöcken unterschiedlicher Struktur (Abb.17). Im *Befehlsblock* stehen in lückenloser Folge alle Befehle. Der *Datenblock* sei derjenige Teil des Programms, der feste Zahlenwerte und Adressen enthält. Beide Blöcke werden beim Ladevorgang vom Maschinenprogramm gefüllt. Daneben umfasse der *Referenzblock* alle weiteren Speicherplätze, auf die sich Befehle des Befehlsblockes beziehen. Darunter fallen

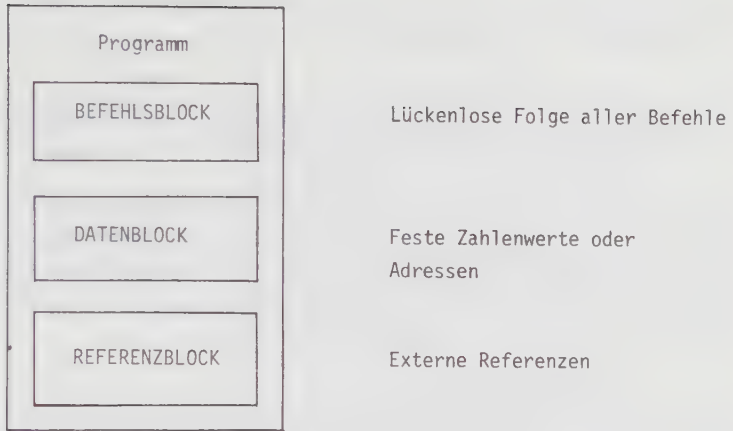


Abb. 17: Prinzipielle Programmstruktur

die im Quellenprogramm für den späteren Ablauf reservierten Speicherplätze (Variablen), sowie Teile anderer Programme, die als externe Referenzen in Form von Unterprogrammen oder Datenfeldern mitbenutzt werden.

Die Verschiebung eines solchen Maschinenprogramms geschieht durch Transport von Befehls- und Datenblock zu dem gewünschten Adreßbereich, sowie Modifizierung beider Blöcke.

Jedes Maschinenprogramm besteht aus einer - möglichst geringen - Anzahl der oben genannten Blockstrukturen, die beliebig aufeinander folgen können. Um die Verschiebung eines Programms zu ermöglichen, sind - wie aus den folgenden Ausführungen hervorgeht - zur Kennzeichnung eines Blockes gewisse Zusatzinformationen in den Code einzuschleusen.

Modifizierung des Programms

Bei dem betrachteten Mikrorechner wird zwischen Ein-, Zwei- und Drei-Byte-Befehlen unterschieden.

Ein-Byte-Befehle enthalten außer dem Operationscode keine weiteren Informationen. Sie brauchen nicht modifiziert werden.

Zwei-Byte-Befehle setzen sich aus dem Operationscode und einem Informationsbyte zusammen. Es wird vereinbart:

Das Informationsbyte eines Zwei-Byte-Befehls enthält keine Adreßbezüge. Es sind nur Daten zugelassen.

Adressen treten nur bei Drei-Byte-Befehlen auf, so daß dann diese zu modifizieren sind. Zusätzlich ist jedoch bei einer bestimmten Befehlsklasse (Load index immediate, LXI-Befehl), auch die Angabe von Daten als Operand möglich. Um im Einzelfall eine Entscheidung treffen zu können, sind zusätzliche Markierungen notwendig.

Für derartige Markierungszwecke wird die Einführung sogenannter *Flag-Befehle* vorgeschlagen [27]. Es handelt sich dabei um Befehle, die für den Programmablauf - abgesehen von der Inkrementierung des Programmzählers - wirkungslos sind, möglichst geringe Ausführungszeit beanspruchen und im "normalen" Programmablauf nicht vorkommen dürfen. Die erforderlichen *Flag-Befehle* werden direkt vom Übersetzer eingefügt.

Definition 7.1

Flag-Befehle dienen zur Kennzeichnung nicht zu modifizierender Folgebefehle oder zur Ankündigung spezieller Blockstrukturen.

Im einzelnen wurden die in Tab. 6 angeführten Flagbefehle festgelegt.

Um dem Binder die Länge eines Datenblockes zu kennzeichnen, vermerkt der Übersetzer nach der formgerechten Ankündigung eines Datenblockes gemäß Tab.6, die Länge des jeweiligen Datenblockes in den folgenden zwei Bytes.

Flag-Befehl	Hinweis
MOV C,C	Ankündigung eines Datenblockes
MOV B,B	Folgend nach MOV C,C: Anschließend Block enthält Adressen Vor einem Drei-Byte-Befehl: Operand kennzeichnet externe Unterprogrammadresse
MOV A,A	Folgend nach MOV C,C: Anschließend Block enthält Konstanten Vor einem Drei-Byte-Befehl: Keine Modifizierung des Operanden

Tab.6: Bedeutung der Flag-Befehle

Die Verschiebung eines so vorbereiteten Maschinenprogramms geschieht nun durch den Transport von Befehls- und Datenblöcken zu dem gewünschten neuen Adreßbereich, sowie der Modifizierung beider Blöcke um die entsprechende Adreßdifferenz (address offset). Einzelheiten zur praktischen Implementierung sind aus [41] zu entnehmen.

7.2 Der Binder (linkage editor)

Die Aufgabe des Binders besteht in der Untersuchung eines vorgegebenen Hauptprogramms auf externe Unterprogrammverzweigungen und dem Anbinden der entsprechenden Bibliotheksunterprogramme an das Hauptprogramm. Erfolgen in einem

Unterprogramm weitere Programmaufrufe, so sind diese Programme ebenfalls anzubinden. Die Arbeitsweise des Binders muß somit eine Mehrfachverschachtelung der Unterprogramme zulassen.

Die Durchführung des Bindevorganges erfordert einige zusätzliche Festlegungen.

- a) Jedes Unterprogramm wird prinzipiell mit einem CALL-Befehl aufgerufen und formal mit dem RETURN-Befehl abgeschlossen. Sprünge zu Unterprogrammen sind durch den entsprechenden Flag-Befehl MOV B,B vorher anzukündigen.
- b) Am Anfang der Bibliotheksprogramme befinden sich jeweils zwei Längenbytes, die die Länge des betrachteten Unterprogramms beinhalten.
Die Bibliotheksprogramme stehen in einem ROM-Bereich des Übersetzersystems. Für andere Zwecke genutzte "Zwischenräume" zwischen einzelnen Bibliotheksbereichen sind ebenfalls durch Längenbytes zu kennzeichnen.

Punkt a) dient zur eindeutigen Kennzeichnung eines eventuell anzubindenden Unterprogramms.

Die Angabe der Längenbytes nach Punkt b) ermöglicht dem Binder bei Arbeitung eines CALL-Befehles die Feststellung, ob eine zulässige Unterprogrammadresse vorliegt. Zulässig sind dabei nur die Anfangsadressen der Unterprogramme. Um nicht eine zusätzliche Tabelle verwenden zu müssen, wird die Bibliothek als gekettete Liste aufgefaßt, bei der die Längenangaben der einzelnen Unterprogramme Kettungszeiger darstellen.

Es ist weiterhin zu berücksichtigen, daß bei Ablauf eines übergeordneten Programms, eventuell mehrmals das gleiche Unterprogramm angesprochen wird. Zur korrekten Bearbeitung derartiger Fälle (nur einmaliger Bindevorgang) wurde eine zusätzliche temporäre Tabelle - hier Bindertabelle genannt - vorgesehen, in der jeweils angesprochene Unterprogramme mit ihrer ursprünglichen Adresse einzutragen sind, sofern dies bisher noch nicht der Fall war. Zusätzlich erfolgt der Eintrag der "Zieladresse", um spätere Referenzen zu ermöglichen.

Auch diejenigen RAM-Speicherplätze, auf die ein Programm mittels der Befehle

LHLD, SHLD, STA, LDA

zugreift [28], sind in dieser Tabelle zu vermerken.

Ein Eintrag in die Bindertabelle besteht aus:

- zwei Bytes zur Eintragung der ursprünglichen Adresse,
- zwei Bytes für die neue Adresse und
- einem zusätzlichen Byte zur Angabe des Typs eines Eintrages (Unterprogramm oder Speicherplatz) bzw. zur Registrierung nach durchgeführtem Bindevorgang.

Prinzip des Bindevorganges

Mit Hilfe des in Kapitel 9 beschriebenen LINK-Kommandos, kann die Anfangsadresse eines zu bindenden Programms festgelegt werden. Vor der Durchführung des eigentlichen Bindevorgangs ist demnach das Hauptprogramm und die eventuell entstehende Adreßdifferenz zu modifizieren. Gleichzeitig erfolgt die Eintragung externer Referenzen des Hauptprogramms in die Bindertabelle.

Der eigentliche Bindevorgang besteht in der Verschiebung aller in der Bindertabelle vermerkten Unterprogramme an das Hauptprogramm und der jeweiligen Adreßmodifizierung der Programme nach Abschnitt 7.1. Werden dabei weitere Unterprogrammaufrufe festgestellt, die noch nicht in der Bindertabelle verzeichnet sind, so ist die jeweilige Adresse in die Tabelle einzutragen und der erforderliche Speicherplatz für das Unterprogramm zu reservieren.

Für externe Referenzen deren Adressen bereits in der Tabelle vermerkt sind, genügt die Übernahme der neuen Zieladresse.

Der Bindevorgang ist dann abgeschlossen, wenn alle in der Bindertabelle aufgeführten Referenzen bearbeitet wurden.

Einzelheiten der Programmtechnischen Realisierung des vorgestellten Konzeptes sind in [41] angegeben.

8. DAS KOPPLUNGSSYSTEM

Bei der Programmerstellung für Aufgaben der Automatisierungs- und Prozeß-rechentechnik sind oft gewisse Randbedingungen zu beachten. Außer der Mög-lichkeit, daß ein Programm fernab von vorgegebenen Automatisierungs- oder Meßdatenverarbeitungsaufgabe - hier kurz Prozeß genannt - erstellt und aus-führlich getestet wird, muß auch z.B. an sogenannte Inbetriebnahmephasen mit den dabei oft unerläßlichen Programmmodifikationen gedacht werden.

Jede Programmmodifikation erfordert eine erneute Übersetzung der Quelle mit anschließendem Test des Programms. Das betrachtete Übersetzersystem unter-stützt durch seine Dialogfähigkeit weitgehend diese Arbeitsweise, wobei die Integration des Übersetzersystems in eine vorhandene Mikrorechnerkonfigu-ration durch die Ankopplung des Übersetzersystems Ü an den zu programmieren-den Rechner R, über jeweils serielle Schnittstellen gemäß Abb.4 ermöglicht wird. Zur Diskussion der Aufgaben des nachfolgend zu erläuternden *Kopplungs-systems* sei diese Abbildung hier nochmals eingefügt.

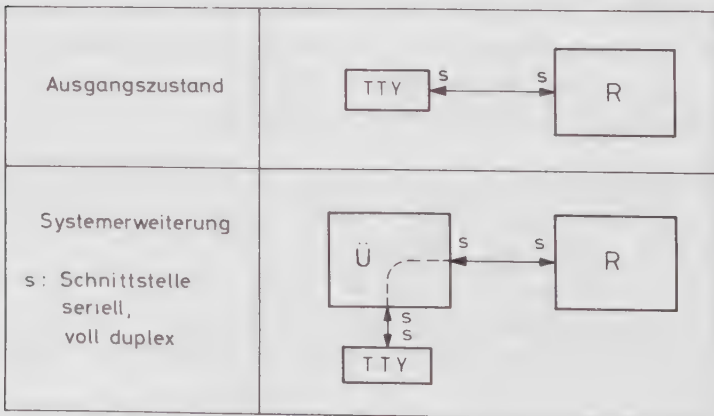


Abb. 4: Prinzip der Ankopplung des Übersetzungssystems Ü an den Rechner R.

Das oben angeführte Konzept weist eine Reihe von Vorteilen gegenüber der herkömmlichen Mikrorechnerprogrammierung auf, so z.B.

- Programmerstellung in einer höheren Programmiersprache direkt bei dem zu programmierenden System (schnelle Zugriffsmöglichkeit) ohne Störung eines eventuellen Prozeßablaufes,
- Interaktiver Programmtest und Korrektur,
- Möglichkeit des sofortigen Programmtransfers in den Rechner R, wodurch nach Abwicklung des Bindevorgangs, ein Programmtest in R bei Verwendung der dort vorhandenen Prozeßinterfaces durchgeführt werden kann,
- einfacher Transport und Anschluß des Übersetzersystems.

Zur Koordination des gewünschten Datenaustausches zwischen dem Rechner R und dem Übersetzungssystem Ü - nachfolgend abgekürzt durch R und Ü - bzw. der entsprechenden Konsolperipherie dient das *Kopplungssystem*, ein Modul des Übersetzersystems.

Gemäß Abb.4 erfolgt jede Eingabe von der Konsole und jede Ausgabe von R zuerst an Ü. Dem Kopplungssystem obliegt nun die Aufgabe,

- a) von der Konsolperipherie eingegebene Zeichen entweder als Übersetzereingabe an die Eingabemodule des eigentlichen Übersetzers weiterzuleiten oder anhand einer Zusatzinformation an R wieder auszugeben,
- b) Ausgabeinformationen von R an die Konsole umzuleiten und
- c) den Transport (Kopie) eines in Ü vorhandenen Speicherbereiches nach R, bei vorausgehender Eingabe eines entsprechenden Kommandos (TRANSFER) abzuwickeln (Programmsicherung).

Es ist zu beachten, daß zur Durchführung dieser Aufgaben bei dem Rechner R keine zusätzliche Soft- oder Hardware implementiert werden soll, sondern nach Ankopplung von Ü über die serielle Schnittstelle, die oben angeführten Möglichkeiten nur durch entsprechende Maßnahmen in Ü zu realisieren sind.

Voraussetzung:

Die betrachteten Systemkomponenten (Konsole, Ü und R) besitzen die gleiche serielle Schnittstelle, so daß eine Kopplung der Systeme nach Abbildung 4 möglich ist. Für die praktische Realisierung wurde von der üblichen V24-Schnittstelle ausgegangen, wobei - von dem Bezugspotential abgesehen - nur die für den Konsolverkehr wichtige Empfangs- und Sendeleitung benötigt wird.

Konzept:

Die eigentliche Übersetzertätigkeit und der Text eines erstellten Programmes innerhalb des Übersetzersystems selbst, sei als "Grundzustand" des Übersetzersystems definiert. Damit der Rechner R auch weiterhin Prozeßmeldungen und Daten auf der Konsole ausgeben sowie Daten von der Konsole übernehmen kann, ist die Möglichkeit der Unterbrechung dieses "Grundzustandes" vorgesehen. Dabei bewirkt jedes von R ausgegebene Zeichen einen Interrupt in Ü (initialisiert durch die Meldung des entsprechenden Transmitter/Receivers: "Daten vorhanden"). Das zugeordnete Unterbrechungsprogramm realisiert dann die Ausgabe des übernommenen Zeichens an die Konsole, das System befindet sich im Zustand "Rechner", der durch den Datenaustausch zwischen Rechner und Konsole gekennzeichnet ist. Der Grundzustand kann wieder durch Eingabe des Steuerzeichens EOT (End of Transmission) erreicht werden.

Wünscht der Programmierer, z.B. zur Eingabe eines Monitorkommandos an R, selbsttätig von dem "Grundzustand" in den Zustand "Rechner" überzugehen, so ist die Eingabe eines weiteren Steuerzeichens -z.B. des Zeichens ACK (Acknowledge) - erforderlich.

Die oben angedeutete Vorgehensweise ermöglicht den Konsolbetrieb des Rechners (Aufgabe a) und b)), obwohl die Konsole mit dem Übersetzungssystem Ü verbunden ist. Zur Realisierung eines eindeutigen Konsolprotokolls wurde bei Änderung des Systemzustandes die Ausgabe der zugeordneten Kennungen

"RECHNER"

bzw.

"ÜBERSETZER"

vorgesehen.

Um ein erstelltes Programm anschließend nach R transferieren und dort starten zu können, müssen folgende Voraussetzungen erfüllt sein:

- Der erforderliche RAM-Speicherbereich steht zur Verfügung,
- in R ist ein Unterprogramm zum Einlesen von Maschinencode nach [13] vorhanden.

Der entsprechende Maschinencode kann durch Eingabe des Kommandos TRANSFER (siehe Kapitel 9.) von Ü nach R übertragen werden. Zu diesem Zweck ist die Ausgabeinformation in Ü gemäß [13] zu formatieren und das Unterprogramm zum Einlesen des Maschinencodes in R zu starten.

Die Software-Module zur Abwicklung der oben genannten Aufgaben sind in dem sogenannten Kopplungssystem zusammengefaßt und sollen dem Anwender die Programmerstellung bzw. die Programmodifikation im Rahmen der Inbetriebnahme-phase erleichtern.

9. DAS KOMMANDOSYSTEM

Die Aufgabe des Kommandosystems besteht in der möglichst umfassenden Unterstützung des Anwenders bei der Programmerstellung, wobei prinzipiell zwischen drei Arbeitsphasen zu unterscheiden ist. Dies sind:

- Die Eingabephase, während der eine Bearbeitung des Quellenprogramms durch den Programmierer erfolgt,
- die Übersetzungsphase, an die sich der Test- und Bindevorgang des erzeugten Maschinencodes anschließen und
- die Phase zur Sicherung der erstellten Programme.

Für alle diese Arbeitsphasen sind zur Einleitung und Spezifizierung eines gewünschten Ablaufes Kommandos erforderlich. In diesem Kapitel sollen die zur Programmerstellung bei Mikrorechnern vorgeschlagenen Kommandos betrachtet werden, wobei die in [48] angeführten Grund-Kommandos auch für Mikrorechner zur Verfügung stehen. Es wird vorwiegend auf die Semantik der Kommandos eingegangen, da deren Syntax unter Punkt 2 der BASEM-Syntax hervorgeht. Bezüglich der Semantik des Kommandos TIME sei auf [48] verwiesen.

Kommandos, deren Implementierung noch nicht abgeschlossen ist, sind durch eine zusätzliche Markierung in der Form "*" vor dem Kommandonamen gekennzeichnet.

9.1 Organisation

Bei der Konzeption des Übersetzersystems war zu beachten, daß die Möglichkeiten bisheriger Mikrorechner-Entwicklungssysteme [13] gewahrt wurden. So gestatten u.a. diese Systeme in einem einfachen Monitor-Betrieb, eine Vielzahl sogenannter Monitorkommandos zur Abwicklung spezifischer Systemleistungen.

Der Übergang von dem genannten Monitor-Betrieb (Grundzustand) zu dem Kommandosystem des Übersetzers erfordert deshalb ebenfalls ein spezielles Monitor-

kommando. Wie in Abb. 18 angedeutet, wurde zu diesem Zweck das Kommando "ⓐBASEM" implementiert. Ebenso wie alle weiteren Kommandos des Übersetzersystems, erfolgt die Einleitung des Kommandos durch das Sonderzeichen "ⓐ".

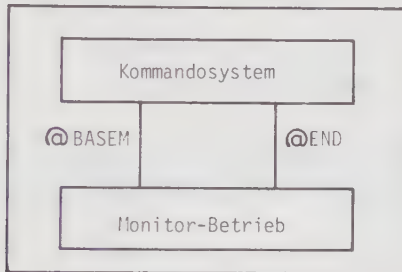


Abb. 18: Aufruf des Kommandosystems

Nach der Meldung des Kommandosystems, steht dem Anwender prinzipiell die Möglichkeit offen, ein neues Programm zu erstellen oder zuvor geschriebenes und als Zwischencode vorliegendes Programm weiter zu bearbeiten. Deshalb erfolgt an erster Stelle die Abfrage "NEW OR OLD ?", auf die nur die Eingabe des entsprechenden Kommandos möglich ist.

NEW: Systemnormierung, Löschung aller Tabellen und des Zwischencodes.

OLD: Weiterbearbeitung von Programmteilen, Tabellen und Zwischencode werden fortgeschrieben.

Erst nachdem diese Anfrage korrekt beantwortet und vom System mit "READY !" quittiert wurde, ist die Eingabe einer beliebigen Quellenzeile oder eines Kommandos - einschließlich der erneuten Eingabe von NEW oder OLD - möglich. Das Kommandosystem ist nun der Analysephase des Übersetzers "untergeordnet" und wird nur aufgerufen, wenn in einer Eingabezeile das "Fluchtsymbol" ⓐ zu identifizieren ist.

Die Rückkehr vom Übersetzer- in den Monitor-Betrieb ermöglicht das Kommando END.

Die Kommandos sind durch ihren Kommandonamen und der jeweiligen Kommandospezifikation gekennzeichnet. Handelt es sich nicht um die Kommandos DUMP, SET oder TIME, so besteht die entsprechende Kommandospezifikation aus maximal drei, durch Kommata getrennte Parameter. Für diese drei Parameter - hier durch die Buchstaben l, m und n gekennzeichnet - gelten folgende Voreinstellungen, wobei Ausnahmen von dieser Vereinbarung explizit bei dem jeweiligen Kommando angegeben sind:

l = 1 ; m = 65 535 (FFFF H); n = 0.

Werden bei einem Kommando der obigen Art keine Parameter angegeben, so erfolgt die Ausführung des Kommandos mit den voreingestellten Werten.

9.2 Aufbereitung von Quellenprogrammen (Editing)

Die Konzeption des Übersetzers erlaubt dem Programmierer durch die weitgehend freie Wahl der Zeilennummer einer einzugebenden Quellzeile, Programmzeilen zu modifizieren bzw. Zeilen einzuschieben. Zusätzlich gestattet nun das Kommandosystem Programmbereiche

- zu löschen (DELETE),
- umzunummerieren (RENUM),
- auf dem LIST-Gerät auszugeben (LIST),
- auf dem PUNCH-Gerät zu stanzen (PUNCH) und
- von dem READ-Gerät zu lesen (READ).

Voraussetzungsgemäß steht für die Programmerstellung als Mindestperipherie nur eine Standard-Fernschreibmaschine zur Verfügung, so daß als LIST - PUNCH - und READ - Gerät diese Fernschreibmaschine bzw. der Anbeustanzer und Anbauleser voreingestellt ist (CONSOL-device).

Sind jedoch weitere I/O-Geräte verfügbar, so kann mit Hilfe des Kommandos ASSIGN diese Voreinstellung entsprechend geändert werden. Das Kommando ASSIGN veranlaßt zu diesem Zweck den Aufruf eines Unterprogramms zur Modifizierung des I/O-Bytes gemäß [12].

Im einzelnen gelten die vorgestellten Kommandospezifikationen.

Kommando	Systemleistung
DELETE 1,m	Löschen der Quellenzeilen 1 bis m.
LIST 1,m	Ausgabe des Quellenprogramms von Zeile 1 bis m auf dem LIST-Gerät.
PUNCH 1,m	Ausstanzen des Quellenprogramms von Zeile 1 bis m auf dem PUNCH-Gerät.
READ 1,m,n	Einlesen eines Quellenprogramms vom READER-Gerät von Zeile 1 bis m. n = 0: Protokoll AUS, nur Protokollie- rung fehlerhafter Zeilen auf dem LIST-Gerät. n = 1: Protokoll EIN.
RENUM 1,m,n	Neunumerierung von Quellenzeile 1 bis m mit Schrittweite n. Voreinstellung: n = 10. Nach Numerierung: Ausgabe von niedrigster und höchster Zeilen-Nummer im neunumerier- ten Intervall.

Tab. 5: Kommandos zur Aufbereitung von Quellenprogrammen.

9.3 Übersetzung, Text und Bindevorgang

Nach Abschluß der Programmeingabe, sollte der Programmierer die Möglichkeit haben, das vorliegende Programm

- zu übersetzen (COMPILE),
- probeweise zu starten (RUN) und
- zu binden (LINK).

Übersetzung eines Quellenprogramms

Zur Übersetzung eines eingegebenen Quellenprogramms in Maschinencode wurde das Kommando COMPILE eingeführt.

COMPILE l,m,n

Codeerzeugung für ein Quellenprogramm von
Quellenzeile l bis m.

Der Parameter n spezifiziert zwischen möglichen
nachfolgend angeführten Testhilfen.

Um den Anwender - besonders in der Testphase - möglichst weitgehend zu unterstützen, mußte ein Konzept zur Realisierung verschiedener Testhilfen erarbeitet werden. Im Rahmen der Übersetzerphase eines Programmes erschien dabei u.a. wichtig vorzusehen:

- Die *TRACE-Vorbereitung* eines Programmes (n=1). Wird ein so vorbereitetes Programm gestartet, besteht bei Abarbeitung des Maschinencodes einer zugeordneten Quellenzeile die Möglichkeit, die Zeilennummer und alle Variablen mit ihrem Namen und aktuellem Wert auszudrucken, an die in dieser Zeile eine Zuweisung erfolgt.
Dem Anwender ermöglicht diese Maßnahme eine schnelle Fehlersuche.
- Die *dynamische Kontrolle* der Indexwerte von Feldern (n=2), wodurch während des folgenden Programmlaufes, eventuelle Überschreitungen von Feldgrenzen besonders vermerkt werden.
- Um spezielle Fehlersuchen auf der Ebene des Maschinenprogramms zu ermöglichen (für Anwender mit weitergehenden Kenntnissen bezüglich der Hardware und Software des zu programmierenden Mikrorechners), besteht die Möglichkeit, den Ausdruck einer *Crossreferenzliste* und *Symboltabelle* (n=3) nach Ablauf der Übersetzungsphase zu veranlassen.

Als Crossreferenzliste sei hier die Aufstellung der Zuordnungen zwischen Zeilennummern des Quellenprogramms und den jeweiligen Anfangsadressen der Maschinencodezeilen verstanden.

In der Symboltabelle sind alle Variablen eines Programms mit ihren Speicheradressen vermerkt.

Ausführung eines übersetzten Programms

Zum Test eines übersetzten Programms ist das Kommando RUN vorgesehen. Im einfachsten Fall besteht das Kommando RUN nur aus dem Kommandonamen ohne Parameterliste.

RUN Ausführung eines vorliegenden Maschinenprogramms.

Auch der abschnittsweise Test eines übersetzten Programms ist möglich, erfordert jedoch, daß das Programm nicht mit "zeilenüberschreitender" Codeoptimierung (siehe Kapitel 6) übersetzt wurde. Um im Normalfall auf diese Codeoptimierung nicht verzichten zu müssen und andererseits im COMPILE-Kommando nicht unnötige viele Spezifikationen vorzusehen wird vorgeschlagen, die Abschaltung der zeilenüberschreitenden Optimierung mit der TRACE-Vorbereitung zu koppeln. Die Spezifikation "TRACE-Vorbereitung" im COMPILE-Kommando führt dann auch zur Abschaltung der erwähnten Optimierung, was jedoch im Fall des ausschließlichen Programmtest vertretbar ist. Um den TRACE-Ausdruck eventuell unterdrücken zu können, muß ein zusätzlicher Parameter im RUN-Kommando vorgesehen werden. Das RUN-Kommando hat dann die folgende Form:

RUN l,m,n Start eines übersetzten Programms von Quellzeile
 1 bis m, bei TRACE-Vorbereitung des Programms im
 COMPILE-Kommando.

 n = 0: Unterdrückung des TRACE-Ausdruckes.

 n = 1: Mit TRACE-Ausdruck.

Eine Kombination der Kommandos COMPILE und RUN in einfachster Form stellt das Kommando START dar.

START Übersetzung des Zwischencodes (gesamtes Programm)
 mit anschließender Ausführung des erzeugten Maschinen-
 programms.

Dieses Kommando soll besonders zur Übersetzung und Test kleinerer Programme dienen.

Bindevorgang (linkage editing)

Nach dem Übersetzungsvorgang sind in dem erzeugten Maschinenprogramm eine Anzahl von Bezügen zu Modulen des Übersetzersystems vorzufinden (z.B. Unterprogrammadressen der Library). Um die Lauffähigkeit des erzeugten Maschinenprogramms auch außerhalb des Übersetzersystems zu sichern, ist deshalb die Durchführung eines Bindevorgangs nach Kapitel 7 erforderlich. Zur Realisierung dieser Aufgabe wurde das Kommando LINK vorgesehen.

- | | |
|--------|--|
| * LINK | Binden eines Maschinenprogramms. |
| l: | Anfangsadresse des zu erzeugenden Objektcodes. |
| m: | Anfangsadresse des gewünschten Variablenbereiches in Schritten von 256 Byte (1/4k Byte). |
| n: | Anfangswert des Stackpointers. |

Der Benutzer kann demnach Objektprogramm, Variable und Stackpointer an die von ihm gewünschte Stelle legen. Ein abschließender Ausdruck des letztlich benötigten Speicherbereiches für das Objektprogramm (erforderlicher ROM-Bereich) und die Variablen (erforderlicher RAM-Bereich), dient zur zusätzlichen Information des Benutzers.

9.4 Sicherung erstellter Programme

Das übersetzte und eventuell gebundene Maschinenprogramm steht zur weiteren Verwendung im RAM-Bereich des Übersetzerprogramms. Um das erstellte Programm vor der Zerstörung zu sichern bzw. das entsprechende Bindeobjekt bei einem anderen System zu implementieren, muß das Programm auf einem nichtflüchtigen Speichermedium "abgelegt" oder direkt zu dem zu programmierenden System transferiert werden.

Kommandos für diese Aufgabe:

- | | |
|------------|--|
| HEXSTANZE | Ausstanzen des vorliegenden Maschineprogrammes auf dem PUNCH-Gerät. |
| * TRANSFER | Transfer des vorliegenden Maschinenprogrammes über eine TTY-Schnittstelle zu anderen Systemen (Kapitel 8). |

Für die Übernahme eines Maschinenprogrammes in PROM's, steht das entsprechende Monitor-Kommando des Entwicklungssystems zur Verfügung [13].

9.5 Weitere Programmierhilfen

Die in diesem Abschnitt angeführten Kommandos sind zur Programmerstellung nicht erforderlich, sollen jedoch den, mit Hard- und Software etwas vertrauteren Anwendern, eine zusätzliche Hilfe in der Programmier- und Testphase bieten. Es werden folgende Kommandos vorgeschlagen:

Programmerstellung

- | | | |
|-----------|-----|--|
| CALCULATE | 1,m | Berechnung der Summe und Differenz von 1 und m.

Die Angabe der Parameter 1 und m ist jeweils in dezimaler oder hexadezimaler Form möglich. Das Ergebnis wird dezimal und hexadezimal ausgedruckt.

Bei Angabe nur des ersten Parameters 1, erfolgt eine Umwandlung von der dezimalen in die hexadezimale Form bzw. umgekehrt. |
| DISPLAY | 1,m | Ausgabe des Speicher-Inhaltes von Adresse 1 bis m. |
| IMMEDIATE | | Anforderung des "immediate modes" (Kapitel 5.1.4) |

Testphase

INFORM Ausgabe der Anfangs- und Endadresse des vorhandenen Maschinenprogramms (Bindeobjektes),

* DUMP X,Y,A(5),... Ausgabe von Variablenwerten.

* SET A = ;, B(5) = 7.532,....
"Setzen" von Variablen.

Natürlich können alle unter dem Begriff "Programmerstellung" eingeordneten Kommandos auch in der "Testphase" angewandt werden. Umgekehrt ist dies jedoch wenig sinnvoll.

LITERATUR

Mikrorechner

- /1/ Davis, S. A Fresh View of Mini- and Microcomputers.
Computer Design, May 1974, S. 67-79
- /2/ Düll, E.H. Der Mikroprozessor in der Steuerungstechnik.
rtp, 11/74, S. 279-285
- /3/ Düll, E.H. Ein frei programmierbares Prozeßsteuergerät
mit Mikrocomputer und PROMS.
Elektronik 1975, Heft 4, S. 93-97,
Heft 5, S. 65-68, Heft 6, S. 101-102
- /4/ Färber, G. Auswirkung der Mikroprozessor-Technik auf
Einsatz und Struktur zukünftiger Prozeßrechen-
anlagen.
Aufsatz aus GFK-GI-GMR, Fachtagung Prozeß-
rechner 1974, Karlsruhe, S. 164-174
Springer-Verlag Berlin 1974
- /5/ Fox, W.F. Einchip 16 but Mikroprozessor
Reyling, G.F. elektronik praxis, Sonderdruck
- /6/ General Automation "the value of micropower".
Inc. 1974 General Automation Inc.
- /7/ Guggenbühl, W. Struktur und Realisierungsformen von Mikro-
computern
Proceedings of the international symposium
and course on mini- and microcomputers and
their Applications, Zürich, June 2-5, 1975,
Acta Press Calgary, Canada
- /8/ Habgood, D. Wie wär's mit einem Mikroprozessor?
Elektronik 1974, Heft 10, S. 379-382

- /9/ Hoffmann, W. Interkama 74: Prozeßdatenverarbeitung.
rtp 1/75, S.3-7
- /10/ Holt, R.M. Current Microcomputer Architecture.
Lemas, M.R. Computer Design, 2/74, S. 65-73
- /11/ Intel Corp. Gewinn durch Microcomputer.
Intel Corp. 1974
- /12/ Intel Corp. Intel 8080 Microcomputer Systems. User's Manual.
Intel Corp., September 1975
- /13/ Intel Corp. INTELLEC 8/MOD 80 Operator's Manual.
Intel Corp., 1975
- /14/ Jacob, H. Struktur von Kleinrechnern.
Vorlesungsskriptum SS 75, Lehrstuhl für
Datenverarbeitung, Ruhr-Universität Bochum
- /15/ Lewandowski, R. Preparation: the key to success with micro-
processors.
Electronics, March 1975, S. 101-106
- /16/ National Mikroprogramm Entwicklungssystem für
Semiconductor NATIONALS Mikroprozessoren.
National Semiconductor, 1975
- /17/ Neunert, H. Systemtechnik mit Mikrocomputern.
Elektronik 1974, Heft 10, S. 391-395
- /18/ Offer, U. Stand und Tendenzen auf dem Gebiet der Prozeß-
rechner Hardware.
Aufsatz aus GFK-GI-GMR, Fachtagung Prozeß-
rechner 1974, Karlsruhe, S. 63-85,
Springer-Verlag Berlin 1974

- /19/ Ogdin, J. Microcomputer Technique.
Euromicro Newsletter, January 1975, S. 43-75
- /20/ Richter, L. Microprogramming and Microprocessing Acti-
vities in Germany.
Euromicro 4/75
- /21/ Sawin, D.H. Microprocessor-Systems.
Euromicro 4/75
- /22/ Schmidt, G. Die Bedeutung der Mikroprozessoren für
Birck, H. die Prozessregelung.
rtp/74, S. 308-315
- /23/ Thomas, A.T. Design Techniques for Microprocessor
Memory Systems.
Computer Design, August 1975, S. 73-78

Mikrorechnersoftware

- /24/ Bartikowski, P. Untersuchung der Befehlsstruktur und des
Befehlsvorrates gängiger Mikrorechner-Systeme
zur Erstellung systemunabhängiger Software.
Diplomarbeit D 5 am Lehrstuhl für Datenver-
arbeitung der Ruhr-Universität Bochum,
Bochum 1975
- /25/ Becker, K.P. Erstellung von Compiler-Modulen für Mikro-
rechner zur Übersetzung von BASEX-Sprach-
elementen in Maschinencode.
Diplomarbeit D 4 am Lehrstuhl für Datenver-
arbeitung der Ruhr-Universität Bochum,
Bochum 1975

- /26/ Eymeren, M. v. Implementierung einer arithmetischen Unterpro-
Heinzel, W. grammbibliothek am Beispiel des Mikrorechner-
Systems INTELLEC 8 MOD 80.
rtp, 4/76
- /27/ Eymeren, M. v. Ein Konzept zur Verschiebung absolut adressier-
Heinzel, W. ter Mikrorechnerprogramme.
rtp, 1976
- /28/ Intel Corp. Programming Manual for the 8080 Microcomputer
System
Intel Corp., 1974, Santa Clara
- /29/ Intel Corp. 8008 and 8080 PL/M Programming Manual.
Intel Corp., 1975, Santa Clara
- /30/ Intel Corp. Users Program Library.
Intel Corp., 1975, Santa Clara
- /31/ Killdall, G.A. High-level language simplifies microcomputer
programming.
Electronics, June 27, 1974, S. 103-109
- /32/ Linhart, R. Erstellung von Compiler-Modulen zur Erzeugung
des Objektcodes.
Diplomarbeit D 13 am Lehrstuhl für Datenver-
arbeitung der Ruhr-Universität Bochum,
Bochum 1976
- /33/ Martinez, R. A Look at Trends in Microprocessor/Microcom-
puter Software Systems.
Computer Design, June 1975
- /34/ Mervill, R.M. Programming a Microprocessor.
Comptes Rendus des Journees d'Electronique,
1974 sur le thème Microprocesseurs, Lausanne

- /35/ National Semiconductor Der Welt erstes Microprocessor "Disk Operating System" (DOS).
Presseinformation, National Semiconductor, 1975
- /36/ Nicoud, J.D. Common Instruction Mnemonics for Microprocessors.
Euromicro Newsletter, April 1975, S.. 22-29
- /37/ Pfaffmann, U. Erweiterung des Unterprogrammes 'PRINT' aus
'LIB I' um 'PRT'.
Interner Bericht Nr. 21 des Lehrstuhls für
Datenverarbeitung der Ruhr-Universität Bochum,
Bochum 1975
- /38/ Schlieper, H.U. Cost and efficiency of the MCS-8008 and MCS-8080
microprocessor-system.
Comptes Rendus des Journees d'Electronique,
1974 sur le thème Microprocesseurs, Lausanne
- /39/ Schmidt, G. Microprocessors/Microcomputers: An Outline
Of Software And Programming.
Proceedings of the international symposium and
course on mini- and microcomputers and their
applications, Zürich, June 2-5, 1975, Acta
Press Calgary, Canada
- /40/ Schneider, P. Software aids for the development of user soft-
ware for microprocessors.
Comptes Rendus des Journees d'Electronique,
1974 sur le thème Microprocesseurs, Lausanne
- /41/ Schoppmeier, B. Konzept und Realisierung von Modulen eines
"linkage editors" für absolut adressierte Pro-
gramme des Mikrorechner-Systems INTELLEC 8 MOD 80.
Studienarbeit S 17 am Lehrstuhl für Datenver-
arbeitung der Ruhr-Universität Bochum, Bochum
1976

- /42/ Nguyen Huu Thuan Erstellung der Software und Hardware zur Implementierung von Systemvariablen am Beispiel des Mikrorechner-Systems INTELLEC 8 MOD 80. Diplomarbeit D 6 am Lehrstuhl für Datenverarbeitung der Ruhr-Universität Bochum, Bochum 1976
- /43/ Wulfinghoff, D.R. Market Factors Portend Design Changes in Computers.
Computer Design, August 1975, S. 81-82

Höhere Programmiersprachen für Prozeßaufgaben

- /44/ Ariger, A. Computerdesign und Systemsicherheit.
Proceedings of the international symposium and course on mini- and microcomputers and their applications, Zürich, June 2-5, 1975, Acta Press Calgary, Canada
- /45/ Dietz BASEX Beschreibung
Computer Systeme Fa. H. Dietz, Mülheim, Januar 1975
- /46/ Fraade, D.J. Probleme beim Einsatz von Prozeßrechnern, Ergebnisse eines Rundtischgespräches.
rtp 2/75, S. 50-55
- /47/ Goldenberg, A. Beschreibung der Prozeßsprache BASEX.
Schlier, Ch. Interner Bericht C 25 des Instituts für
Schupp, W. Physik der Universität Freiburg
- /48/ Goldenberg, A. Höhere Prozeßsprachen für kleinere Rechner
Schlier, Ch. - Das Beispiel BASEX.
Schupp, W. Aufsatz aus GFK-GI-GMR, Fachtagung Prozeßrechner 1974, Karlsruhe, S. 436-445,
Springer-Verlag Berlin 1974
- /49/ Huppertz, H.O. Die Programmiersprache CORAL 66.
rtp 7/75

- /50/ Janecke, R.
Löns, K. AUTRAN, ein Softwaresystem für die Prozeß-
steuerung.
rtp 10/75, S. 320-324
- /51/ Lümke mann, H. Heutiger Entwicklungsstand der Kompaktrechner-
Software.
rtp 4/75, S. 101-110
- /52/ Martinez, R. A Look at Trends in Microprocessor/Microcom-
puter Software Systems.
Computer Design, June 1975
- /53/ Mittendorf, H. Higher-Level Languages for the Programming
of Process Computers.
Siemens Forsch.-Entwickl. Ber., (Bd. 3/1974),
Nr. 5
- /54/ Oitzl, E. Kleinrechner für industrielle Automatisierungs-
aufgaben.
rtp 3/75, S. 76-81
- /55/ Paul, M.
Schuler, H. Marktübersicht: Prozeßrechner 75
rtp 10/75, S. 297-305
- /56/ Paul, W.
Zielstorff, A.W. Erfahrung mit der Benutzung eines Prozeß-
FORTRAN.
rtp 5/75, S. 151-156
- /57/ Platz, G. Interaktives Programmieren.
Online Juli/Aug. 1975, S. 489
- /58/ Purdue Europe Report on Real-Time Extensions and Implemen-
tation of Real-Time BASIC.
Purdue Europe, Real-Time BASIC Committee
Brüssel, First Report Oct. 1975
- /59/ Scheytt, W. Minicomputer.
Elektronik 10/75, S. 82-90

- /60/ Schultz, J.T. Real-Time Multi-User BASIC.
Hewlett-Packard Journal, January 1976,
Volume 27, Number 5
- /61/ Thane, H.W. FPL, Programmiersprache für Computer-Muffel.
rtp 12/74, S. 329, 330
- /62/ Wagner, F. Process BASIC - Ein Programmiersystem für
Woda, H. Prozeßlenkung mit Kleinrechnern.
Aufsatz aus GFK-GI-GMR, Fachtagung Prozeß-
rechner 1974, Karlsruhe, S. 425-435,
Springer-Verlag Berlin 1974
- /63/ Wagner, F. PROCESS-BASIC.
Woda, H. Gesellschaft für Kernforschung mbH,
Karlsruhe, August 1975
- /64/ Winkler, H. BASIC - eine dialogorientierte anwenderfreund-
liche Programmiersprache.
Elektronik 10/75, S. 99-102

Theorie der Datenverarbeitung und Übersetzer

- /65/ Abramson, H. Theory and Application of a Bottom-Up Syntax-
Directed Translator.
ACADEMIC PRESS, New York, 1973
- /66/ Aho, A.V. The Theory of Parsing, Translation and Compiling.
Ullmann, J.D. Volume I: Parsing.
Prentice-Hall, Inc., London 1972
- /67/ Aho, A.V. The Theory of Parsing, Translation and Compiling.
Ullmann, J.D. Volume II: Compiling.
Prentice-Hall, Inc., London 1973
- /68/ Arnold, G. Neue Aspekte zur Entscheidungstabellentechnik.
Halbach, H.D. Elektronische Rechenanlagen, 6/75, S. 282-286

- /69/ Cocke, J. Programming Languages and Their Compilers.
Schwartz, J.T. New York University, 1970, Courant Institute
of Mathematical Sciences
- /70/ Conway, M.E. Design of a separable transition diagram
compiler.
CACM 6 (1963), 396 ff
- /71/ Dotzauer, E. Grundlagen der Datenverarbeitung, Teil 1 und 2.
Carl Hanser Verlag, München 1968 bzw. 1971
- /72/ ECMA Standard ECMA European Computer Manufactures
Association for Minimal BASIC.
Third Draft, April 1975
- /73/ Forster, J.M. Automatische Syntax-Analyse.
Carl Hanser Verlag, München 1971
- /74/ Gries, D. Compiler Construction for Digital Computers.
John Wiley, London 1971
- /75/ Gesellschaft für Zweite Fachtagung über Programmiersprachen,
Informatik Saarbrücken, März 1972, Bericht Nr. 4
- /76/ Haas, P. Übersetzer für elektronische Rechenautomaten.
Markowski, H. Verlag Moderne Industrie, München 1971
- /77/ Healstead, M.H. A Laboratory Manual für Compiler and Operating
System Implementation.
American Elsevier Publishing Co., Inc., 1974
- /78/ Heller, G. Aspekte zur Auswahl von Prozeßrechnern.
rtp 4/69, S. 139-146
- /79/ Heyderhoff, P. Informationsstrukturen.
Hildebrand T. BI, Zürich 1973

- /80/ Hopgood, F.R.A. Compiler.
Carl Hanser Verlag München, 1970
- /81/ Kley, A. Was ist ein rekursiver Abstieg?
Jonsson, B. Elektronische Rechenanlagen, Heft 4, 1975,
S. 180-188
- /82/ Knöpker, R. Entwicklung rekursiver endlicher Akzeptoren.
Diplomarbeit PÜ-S14, TH Darmstadt, Fach-
bereich Informatik
- /83/ Knuth, D.E. Top-down syntax analysis.
Acta Informatica 1 (1971), S. 79-110
- /84/ Knuth, D.E. Syntactical Algorithmus.
Addison-Wesley, London
- /85/ Knuth, D.E. Compilers.
Addison-Wesley, London
- /86/ Kupka, I. Dialogsprachen.
Wilsing, N. Verlag B.G. Teubner, Stuttgart 1975
- /87/ Kurtz, T.E. Proposed standard for Minimal BASIC.
Committee Correspondence of American National
Standard Institute (ANSI), Washington, Dec. 75
- /88/ Lee, J.A.N. Computer Semantics.
Van Nostrand Reinhold Company, Toronto 1972
- /89/ Lee, J.A.N. The Anatomy of a Compiler.
Van Nostrand Reinhold Company, New York 1974
- /90/ Menzel, W. Formale Sprachen.
Vorlesungsskriptum der Fakultät für Informatik,
Universität Karlsruhe 1973

- /91/ Rehbein, H. BASIC - leicht gemacht.
VDI Verlag GmbH, Düsseldorf 1972
- /92/ Schneider, H.J. Compiler, Aufbau und Wirkungsweise.
Walter de Gruyter Verlag, Berlin 1975
- /93/ Seegmüller, G. Einführung in die Systemprogrammierung.
BI Mannheim 1974
- /94/ Waldschmidt, H. Optimierungsfragen im Compilerbau.
Carl Hanser Verlag, München 1974
- /95/ Weingarten, F.W. Translation of Computer Languages.
Holden-Day, Inc., San Fransisco 1973
- /96/ Wettstein, H. Systemprogrammierung.
Carl Hanser Verlag, München 1972
- /97/ Weaver, A.C. A BASIC language interpreter for the INTEL
Tindall, W.H. 8008 microprocessor.
Danielson, R.L. University Illinois, June 1974,
PB-235874/5

ANHANG

A 1: Fehlercodes der Analysephase

<u>Code</u>	<u>Bedeutung</u>
01	Zeilennummer fehlt
02	Zeilennummer ist zu groß
03	Zeile endet nach der Zeilennummer
04	BASEM-Anweisung fehlt oder ist unzulässig
05	Zeile ist zu lang (mehr als 69 Zeichen)
06	Zeile enthält zu viele Fehler
07	Zeilenende vorzeitig gefunden (unvollständig)
08	Information nach Anweisungsende
10	Name mehrfach definiert
11	Variablenname fehlt
12	Variablenname ist unzulässig
13	Feldvereinbarung ist unzulässig
14	Zeichen ist unzulässig; Komma fehlt
15	Teilstrings sind nicht erlaubt
16	Marke fehlt oder ist unzulässig
17	Laufparameter muß einfache Variable sein
20	Element noch nicht implementiert
21	Fehler in Klammerstruktur
22	Operator hier unzulässig
23	Aufbau des Ausdrucks unzulässig
24	Index- oder Parameteranzahl ist falsch
25	Zahlenkonstante falsch oder unzulässig
26	Keine Zuweisung im LET
27	Reserviertes Wort hier unzulässig
28	Komma hier unzulässig
A1	Abbruch wegen Speichermangel
A2	Abbruch wegen Überlauf der Symboltabelle
A3	Abbruch wegen Überlauf der Sprungzieltabelle

A 2: Fehlercodes der Generierungsphase

<u>Code</u>	<u>Bedeutung</u>
C0	Leistung noch nicht implementiert
C1	Marke in der Sprungzieltabelle mehrfach vorhanden
C2	Unbekanntes Sprungziel
C3	Zu viele Feldelemente in einer DIM-Anweisung
C4	Mixed-mode Fehler oder Länge einer Stringvariablen für numerische Zuweisung zu klein
C5	Stacküberlauf (z.B. Operatoren-, Operanden-Stack)
C6	Unzulässige Schachtelung von FOR-Schleifen oder FOR-Anweisung ohne NEXT

Punkt 1.1: Schriftbild des aus dem Zwischencode zurückgewonnenen Quellenprogramms

```

@LIST
10 REM      ***LINEARES GLEICHUNGSSYSTEM MIT ZWEI UNBEKANNTEN***
20 REM
30 REM      A*X1 + E*X2 = C
40 REM      E*X1 + F*X2 = G
50 REM
60 REM      LOESUNG MIT HILFE DER KRAMERSCHEN REGEL
70 REM
75 INPUT "A=",A,"B=",B,"C=",C,"E=",E,"F=",F,"G=",G
80 LET D=A*F-E*E
90 IF D= 0.0000 THEN 140
100 LET X1=(C*F-G*E)/D
110 LET X2=(A*G-E*C)/D
120 PRINT "X1 =",X1," X2 =",X2
125 PRINT "KONTROLLE: C =",A*X1+E*X2," G =",E*X1+F*X2
130 GOTO 150
140 PRINT "DETERMINANTE = 0"
150 END

```

Punkt 1.2: Übersetzung mit Ausdruck von Symboltabelle und Crossreferenz

```

@COM,,3
D.....5CFDH
A.....5CFAM
F.....5CF7H
E.....5CF4H
B.....5CF1H
X1.....5CEEH
C.....5CEDH
G.....5CE3H
X2.....5CE5H
10 =D204H 20 =D204H 30 =D204H 40 =D204H 50 =D204H
60 =D204H 70 =D204H 75 =D204H 80 =D20BH 90 =D2A7H

100 =D2E3H 110 =D2D2H 120 =D2F2H 125 =D326H 130 =D392H
140 =D395H 150 =D3E7H

ISA D200H D3E7H
VSA 5CDFH 5CFFH

```

Punkt 1.3: Anwendungsbeispiel für das Kommando "CALCULATE"

```

@CA 0D3E7H,0D200H
42423 A5E7H
439 01E7H

```

Punkt 1.4: Test des übersetzten Programms

```

@RUN
A=4 B=50 C=100 E=1 F=10 G=50

X1 = 150.00 X2 =-10.000
KONTROLLE: C = 100.00 G = 50.000
READY!

```

```

@RUN
A=443 E=625 C=762 E=310 F=310 G=19
X1 =-3.9760 X2 = 4.0373
KONTROLLE: C = 762.00 C = 19.000
READY!

```

Beispiel 2:

Dieses Beispiel wurde aus einer Arbeit am Lehrstuhl für Informations - technik der TU Berlin zum Test eines BASIC - Interpretierers übernom - men.

Das Beispiel dient hier nicht zur Überprüfung spezifischer Eigenschaften des Übersetzers, sondern soll nur in die Bedienung des Systems einführen.

Das Programm soll aus N Punktepaaren (hier N=5) die Koeffizienten eines Polynoms M-ten Grades (hier M=2) berechnen, so daß die mittlere quadra - tische Abweichung an den Orten der Punktepaare möglichst klein ist

Die exakte Lösung fuer die in dem Beispiel eingegebenen Punktepaare wäre für das Polynom

$$Y = A_0 + A_1 \cdot X + A_2 \cdot X^2$$

$$A_0 = 2, \quad A_1 = 4, \quad A_2 = 1.$$

Punkt 2.1: Schriftbild.

Punkt 2.2: Wirkung des Kommandos RENUM.

Punkt 2.3: Übersetzung.

Für die Übersetzung dieses Programms von 91 Zeilen wird ein Maschinencode von 1826 Bytes angelegt. Dies entspricht im Mit - tel 20 Bytes/Zeile.

Punkt 2.4: Bestimmung des erzeugten Mascinencodebereiches.

Punkt 2.5: Programmtest.

Punkt 2.6: Übersetzung ohne Optimierung.

Für die Übersetzung dieses Programms von 91 Zeilen wird ein Maschinencode von 2187 Bytes angelegt. Dies entspricht im Mit - tel 24 Bytes/Zeile.

Beispiel 2:

Punkt 2.1:

CLIST

```

5 REM      LEAST SQUARE FITTING
7 DIM      K(20), Y(20), A(11,11), B(11)
10 DIM     C(11), P(20)
12 REM     I/O-DEVICE
15 PRINT   DEV ( 4.0000 )
16 INPUT   "POLYNOMGRAD M =",M
17 PRINT
20 FOR I= 1.0000 TO 20.000
22 REM     PUNKTE-PAARE X,Y
25 INPUT   "X =",X(I)," Y =",Y(I)
27 REM     ENDE DER DATEN M=0, Y=0
30 IF X(I)= 0.0000 THEN 43
35 NEXT I
40 STOP
42 REM     AUFBAU DES GLEICHUNGSSYSTEMS
43 LET NUM=I- 1.0000
45 LET MX2=M* 2.0000
50 FOR I= 1.0000 TO MX2
55 LET P(I)= 0.0000
60 FOR J= 1.0000 TO NUM
65 LET P(I)=P(I)+X(J)**I
70 NEXT J
75 NEXT I
80 LET N=M+ 1.0000
85 FOR I= 1.0000 TO N
90 FOR J= 1.0000 TO N
95 LET K=I+J- 2.0000
100 IF K= 0.0000 THEN 115
102 REM     LINKE SEITE
105 LET A(I,J)=P(K)
110 GOTO 120
115 LET A( 1.0000 , 1.0000 )=NUM
120 NEXT J
125 NEXT I
130 LET B( 1.0000 )= 0.0000
131 FOR J= 1.0000 TO NUM
140 LET B( 1.0000 )=B( 1.0000 )+Y(J)
145 NEXT J
150 FOR I= 2.0000 TO N
155 LET B(I)= 0.0000
160 FOR J= 1.0000 TO NUM
162 REM     RECHTE SEITE
165 LET B(I)=B(I)+Y(J)*X(J)**(I- 1.0000 )
170 NEXT J
175 NEXT I
177 REM     UNORDNEN DER REIHENUM RUNDUNGS-FEHLER
178 REM     KLEIN ZU HALTEN
180 LET NM1=N- 1.0000
185 FOR K= 1.0000 TO NM1
190 LET KP1=K+ 1.0000
195 LET L=K
200 FOR I=KP1 TO N
205 IF ABS(A(I,K))<=ABS(A(L,K)) THEN 215
210 LET L=I
215 NEXT I
220 IF L=K THEN 265
225 FOR J=K TO N
230 LET TEMP=A(K,J)
235 LET A(K,J)=A(L,J)
240 LET A(L,J)=TEMP
245 NEXT J

```

```

250 LET TEMP=B(K)
255 LET B(K)=B(L)
260 LET B(L)=TEMP
263 REM          GAUSS ALGORITHMUS
265 FOR I=KPI TO N
270 LET FACT=A(I,K)/A(K,K)
275 LET A(I,K)= 0.0000
280 FOR J=KPI TO N
285 LET A(I,J)=A(I,J)-FACT*A(K,J)
290 NEXT J
295 LET B(I)=B(I)-FACT*B(K)
300 NEXT I
305 NEXT K
310 LET C(N)=B(N)/A(N,N)
325 LET I=INI
330 LET SUM= 0.0000
333 LET IPI=I+ 1.0000
335 FOR J=IPI TO N
340 LET SUM=SUM+A(I,J)*C(J)
345 NEXT J
350 LET C(I)=(B(I)-SUM)/A(I,I)
355 LET I=I- 1.0000
356 IF I>= 1.0000 THEN 330
357 PRINT
358 PRINT "ERGEBNIS"
359 PRINT
360 FOR I= 1.0000 TO N
365 PRINT I,C(I)
370 NEXT I
380 END

```

ERENUM

10

910

Punkt 2.2:

QLIST

```

10 REM          LEAST SQUARE FITTING
20 DIM          X(20), Y(20), A(11,11), B(11)
30 DIM          C(11), P(20)
40 REM          I/O - DEVICE
50 PRINT DEV ( 4.0000 )
60 INPUT "POLYNOMGRAD M =",M
70 PRINT
80 FOR I= 1.0000 TO 20.000
90 REM          PUNKTE-PAARE X,Y
100 INPUT "X =",X(I)," Y =",Y(I)
110 REM          ENDE DER DATEN X=0, Y=0
120 IF X(I)= 0.0000 THEN 160
130 NEXT I
140 STOP
150 REM          AUFBAU DES GLEICHUNGSSYSTEMS
160 LET NUM=I- 1.0000
170 LET MM2=M* 2.0000
180 FOR I= 1.0000 TO MM2
190 LET P(I)= 0.0000
200 FOR J= 1.0000 TO NUM

```

```
210 LET P(I)=P(I)+K(J)**I
220 NEXT J
230 NEXT I
240 LET N=N+ 1.0000
250 FOR I= 1.0000 TO N
260 FOR J= 1.0000 TO N
270 LET K=I+J- 2.0000
280 IF K= 0.0000 THEN 320
290 REM LINKE SEITE
300 LET A(I,J)=P(K)
310 GOTO 330
320 LET A( 1.0000 , 1.0000 )=NUM
330 NEXT J
340 NEXT I
350 LET B( 1.0000 )= 0.0000
360 FOR J= 1.0000 TO NUM
370 LET B( 1.0000 )=B( 1.0000 )+Y(J)
380 NEXT J
390 FOR I= 2.0000 TO N
400 LET B(I)= 0.0000
410 FOR J= 1.0000 TO NUM
420 REM RECHTE SEITE
430 LET B(I)=B(I)+Y(J)*K(J)**(I- 1.0000 )
440 NEXT J
450 NEXT I
460 REM UMORDNEN DER REIHENUM RUNDUNGS-FEHLER
470 REM KLEIN ZU HALTEN
480 LET NM1=N- 1.0000
490 FOR K= 1.0000 TO NM1
500 LET KP1=K+ 1.0000
510 LET L=K
520 FOR I=KP1 TO N
530 IF ABS(A(I,K))<=ABS(A(L,K)) THEN 550
540 LET L=I
550 NEXT I
560 IF L=K THEN 660
570 FOR J=K TO N
580 LET TEMP=A(K,J)
590 LET A(K,J)=A(L,J)
600 LET A(L,J)=TEMP
610 NEXT J
620 LET TEMP=B(K)
630 LET B(K)=B(L)
640 LET B(L)=TEMP
650 REM GAUSS ALGORITHMUS
660 FOR I=KP1 TO N
670 LET FACT=A(I,K)/A(K,K)
680 LET A(I,K)= 0.0000
690 FOR J=KP1 TO N
700 LET A(I,J)=A(I,J)-FACT*A(K,J)
710 NEXT J
720 LET B(I)=B(I)-FACT*B(K)
730 NEXT I
740 NEXT K
750 LET C(N)=B(N)/A(N,N)
760 LET I=NM1
770 LET SUM= 0.0000
780 LET IP1=I+ 1.0000
790 FOR J=IP1 TO N
800 LET SUM=SUM+A(I,J)*C(J)
```

```

310 NEXT J
320 LET C(I)=(B(I)-SUM)/A(I,I)
330 LET I=I- 1.0000
340 IF I>= 1.0000 THEN 770
350 PRINT
360 PRINT "ERGEBNIS"
370 PRINT
380 FOR I= 1.0000 TO N
390 PRINT I,C(I)
900 NEXT I
910 END

```

CCOMPILE

Punkt 2.3:

```

ISA D200H D222H
VSA 571EH 59FFH

```

```

CCA 0D922H,0D200H
43010 1122H
1326 0722H

```

Punkt 2.4:

```

@RUN
POLYNOMGRAD N =2
X =1, Y =7,X =2, Y =14,X =3, Y =23,X =4, Y =34,X =5, Y =47,X =0, Y =0
ERGEBNIS

1.0000 2.0432
2.0000 3.9623
3.0000 1.0059

```

Punkt 2.5:

READY!

CCOM,,80H

Punkt 2.6:

```

ISA D200H DA3BH
VSA 571EH 59FFH

```

```

CCA 0DA3BH,D200H
44171 AC3EH
2137 033EH

```

```

@RUN
POLYNOMGRAD N =2
X =1, Y =7,X =2, Y =14,X =3, Y =23,X =4, Y =34,X =5, Y =47,X =0, Y =0
ERGEBNIS

1.0000 2.0452
2.0000 3.9623
3.0000 1.0059

```

READY!

Lebenslauf

- 1946 geboren am 15. Mai 1946 als Sohn des Ernst Heinzel
und dessen Ehefrau Elfriede Heinzel geb. Nitzschke
- 1953 - 1961 Volksschule in Enkenbach und Kaiserslautern
- 1961 - 1964 Lehre als Fernmeldemonteur
- 1962 - 1965 Berufsaufbauschule (Abendschule)
- 1965 - 1968 Studium der Allgemeinen Elektrotechnik an der
Staatlichen Ingenieurschule für Maschinenwesen,
Kaiserslautern.
Ingenieurprüfung am 5. Februar 1968
- 1968 - 1972 Studium der Elektrotechnik (Regelungs- und Daten-
technik) an der Technischen Hochschule Darmstadt.
Diplomprüfung am 2. Mai 1972
- 1972 - 1973 Wissenschaftlicher Assistent am Lehrstuhl für
Regelungstechnik der Universität Kaiserslautern
(Prof. Dr.-Ing. W. Weber)
- 1973 Eheschließung mit Christine Heinzel geb. Krauleidies
am 21. September 1973
- 1973 - Wissenschaftlicher Assistent am Lehrstuhl für Daten-
verarbeitung der Ruhr-Universität Bochum (Prof. Dr.-Ing.
W. Weber) seit 1. Oktober 1973

Diese Arbeit entstand in den Jahren 1974 bis 1976 am Lehrstuhl für Datenverarbeitung der Ruhr-Universität Bochum.

Herrn Prof.Dr.-Ing. Wolfgang Weber möchte ich für die Anregung, sein förderndes Interesse und die Möglichkeit, dieses Gebiet der Mikrorechen-technik zu bearbeiten, herzlich danken.

Mein Dank gilt ebenso Herrn Prof.Dr.-Ing. Friedhelm Depping für die Übernahme des Korreferates und seine Anregungen zur Gestaltung der Arbeit.

Bochum, Juni 1976

Werner Heinzel

